UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale in Applicazioni Telematiche

A Group Communication Service for OMA Lightweight M2M

Anno Accademico 2013/2014

relatore
Ch.mo prof. Simon Pietro Romano

correlatore
Jaime Jiménez Bolonio

candidato
Domenico D'Ambrosio
matr. M63/229

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Machine-to-Machine (M2M) and Internet of Things (IoT) communications are made possible by the existance of devices (such as sensors) that are capable of measuring the environment and create events. These events are often sent over a network where other machines/devices will process the data from those events, eventually producing output for other machines or ready to be interpreted by a human user.

To manage and configure these devices, in order to obtain the best results for the systems they belong, management protocols are used. Different management protocols which provide similar functionalities exist but, since they were designed to satisfy different requirements, they operates in differently one from another.

Lightweight M2M is a management protocol that was developed in order to address the particular needs and requirements of managing constraint devices, typical of M2M and IoT systems, more efficiently than other management protocol (like SNMP or TR-069).

However LWM2M lacks functionalities like group management, aggregation and network optimization, which could be extremely important when managing an huge number of devices that need to perform the same task and/or have the same properties, thus must be configured and managed in the same way. For this reason we propose a Group Communication Service for LWM2M that provides these features and we evaluate its performances.

In the rest of this chapter we first introduce the problem we are trying to solve, we present the contributions of this thesis project and finally we describe the organization of this thesis.

1.2 Problem Description

LWM2M is a protocol for managing constraint devices, and it is designed to be a one-to-one communication protocol between a LWM2M Server and a LWM2M Client, not many-to-many. Since the server could also manage millions of M2M/IoT devices, aspect such as group management, aggregation and network optimization are of great importance and could produce a great improvement of performance but the protocol does not provide any of these features.
Constrained devices can be large in numbers, but are often related to each other by properties (like location) or functionality. If information needs to be sent (or received from) a group of devices, the use of grouping techniques can reduce latency and bandwidth requirements.

In this thesis we try to solve this lack proposing a solution that implements a group communication service. We introduce a LWM2M Proxy into the system simplifying the design of the management logic in both the client and the server. The proxy will be responsible for forwarding the commands sent by the LWM2M Server to group members as well as aggregating the subsequent responses from LWM2M Clients. The Proxy will also provide functionality for group management and information reporting and provides information to the manager about the connected M2M devices.

1.3 Contribution

The contributions of this thesis includes introduction of a LWM2M Proxy in order to provide a Group Communication Service in LWM2M protocol and increase network performance, a justification for introducing a proxy in LWM2M architecture, a design for such a proxy, a working prototype with minimum functionality, and an evaluation of this prototype.

1.4 Thesis Organization

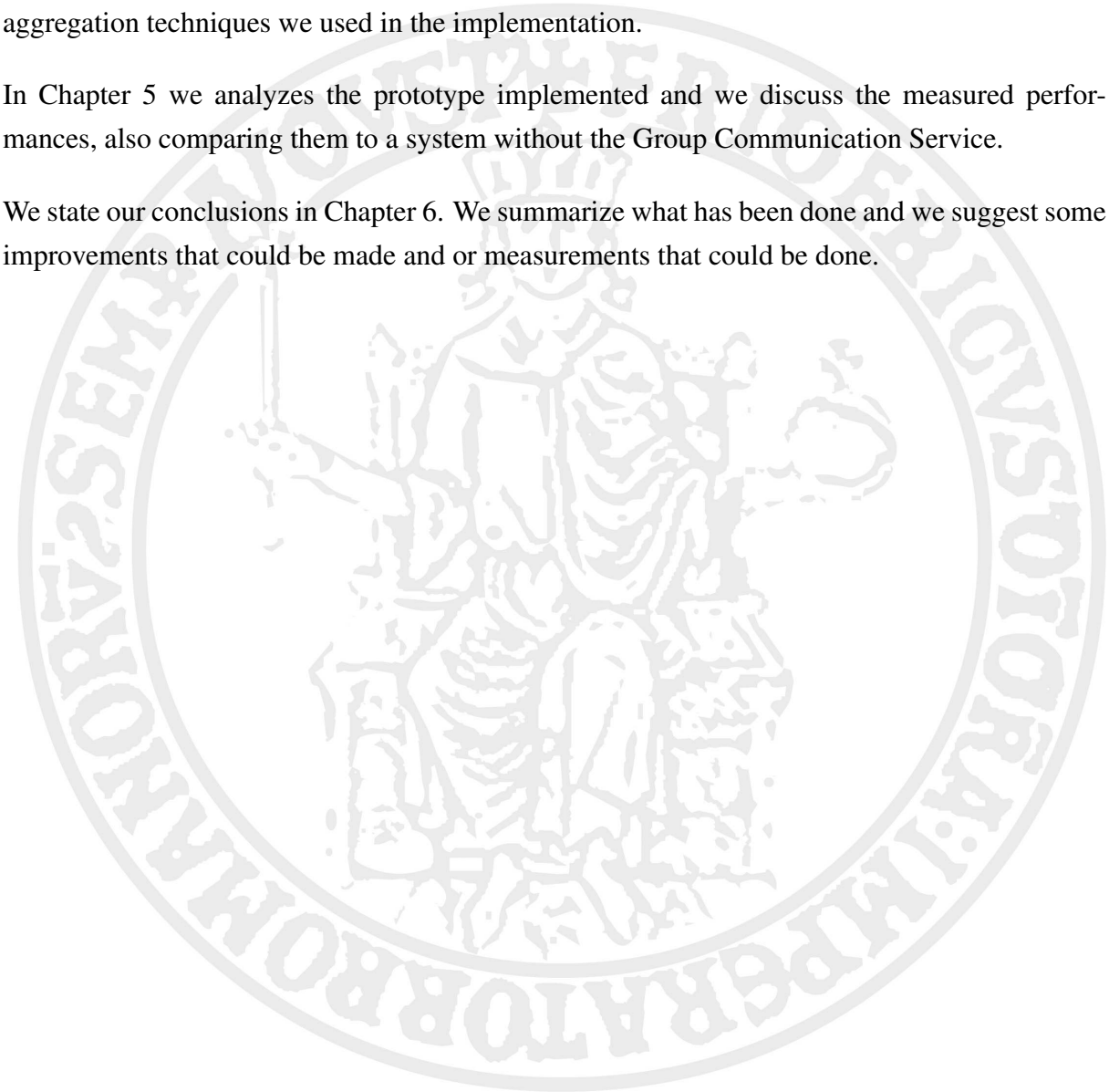
Chapter 2 introduce concept related to this thesis. An explenation of Machine to Machine (M2M) and Internet of Things (IoT) is given in Section 2.1. In Section 2.2, we talk about communication models and mechanisms like REST, Pub/Sub and Multicast. The application protocol CoAP, designed for M2M and IoT devices, is discussed in Section 2.3; Lastly, we talk about managament protocols and introduce SNMP (2.4.1), TR-069 (2.4.2) and the new Lightweight M2M (2.4.3).

In Chapter 3 we explain the decisions we made in the design process. First we list what objectives and requirements we want to satisfy with this work and then we explain the architecture and how the Group Communication Service could work in LWM2M.

In Chapter 4 we describes the implementation of a prototype, including specific software and aggregation techniques we used in the implementation.

In Chapter 5 we analyzes the prototype implemented and we discuss the measured performances, also comparing them to a system without the Group Communication Service.

We state our conclusions in Chapter 6. We summarize what has been done and we suggest some improvements that could be made and or measurements that could be done.



Chapter 2

Background

2.1 M2M and IoT

The term Machine-to-Machine (M2M) is commonly used, in literature and the ICT, to refer to "technologies that allow communication between machines without human intervention" [5, 6, 7]. This type of communication is based on the idea that interconnecting machines together and program them to automatically perform certain actions based on specific events received from other machine can be more useful than having a single machine performing a single task.
M2M communications are made possible by the existance of devices (such as sensors) that are capable of measuring the environment and create events. These events are often sent over a network where other machines/devices will process the data from those events, eventually producing output for other machines or ready to be interpreted by a human user.

Another concept partly overlapping with M2M is the Internet of Things (IoT). The Internet of Things is defined as "things having identities and virtual personalities operating in smart spaces using intelligent interfaces to connect and communicate within social, environmental, and user contexts" [8]. IoT focuses on interconnecting various devices with different size, information processing power and mobility, together to a large area network, usually running on IP [9]. The basic idea of IoT is that "things" such as tag readers, sensors, actuators, mobile phones, and so on, through agreed communication protocols, can interact and cooperate with each other, in order to perform certain common tasks together [10].

M2M and IoT, being broad concepts, can be applied to a big number scenarios [11], some examples are: home and industry automation, healthcare, security, flood control systems, smart grid, monitor and control of buildings temperature, lighting levels and security, vending machines,

automobiles and tracking. The large amount of possible uses of M2M and IoT is the reason why they are belived to provide growing benefit. According to Ericsson and other IT Company, by 2020 there will be 50 billion connected devices [12, 13]. Cisco has estimated that nowdays there are already 13 billion connections [14], this makes a shift of focus from person-to-person communication to machine-to-machine communication to be expected.
Since protocols used for M2M communications vary based on the industry and most of proprietary vertical M2M solution have difficulty scaling [15], multiple standards developing organization, such as 3GPP, ETSI, IEEE and telecommunications industry association (TIA), have begun the process to define network architectures and function to support the unique features of M2M communications in their standard bodies [16].

Future devices to be part of M2M and IoT systems will have more requirements then current ones. For instance as listed by ETSI [17], general requirements like message delivery for sleeping devices, scalability, trusted application and mobility, will be central, but also some more specific ones such as monitoring, connectivity testing, configuration management, grouping capabilities, authentication and data integrity [18]. These are the needs for device management and application management, needs that protocols like SNMP, TR069 and LWM2M specifically address.

2.2 Communication Models and Mechanisms in M2M/IoT

M2M system designers take advantage of many communication models and mechanism to satisfy their requirements. Following we introduce some of the most common used to satisfy M2M and IoT special requirements.

2.2.1 REST

REST is the acronym for *REpresentational State Transfer* defined by Roy T. Fielding in [19]. It is a set of principles that abstracts architectural elements in a distributed hypermedia system in order to design network application. The main characteristics of REST can be summarized in: a stateless client/server protocol; an uniform interface (the overall system architecture is simplified); use of hypermedia; a universal syntax for addressing (URI); self-descriptive messages (the information in the message are all what is needed to understand the request).

Fielding explains that "REST ignores components implementation and protocol syntax in order to focus on the roles of components, the interaction's constraints, and their interpretation of significant data elements". Components in this architecture perform actions on resources using its

representation. For instance, when visiting a web page with a browser, a request for a resource is made to a server, the server transfers the current state of a Web page to the browser. Once the resource is received, the representation (the data) can be rendered.
As explained in [20], in the same way as HTTP, the REST architecture too is based on a sequence of request and response between client and server. In contrast to other systems, such as SOAP [21], where users can define their own methods and resource identifiers, REST requests use HTTP CRUD operations (Create/Read/Update/Delete) to access the repesentational state of a resource identified by an URI.

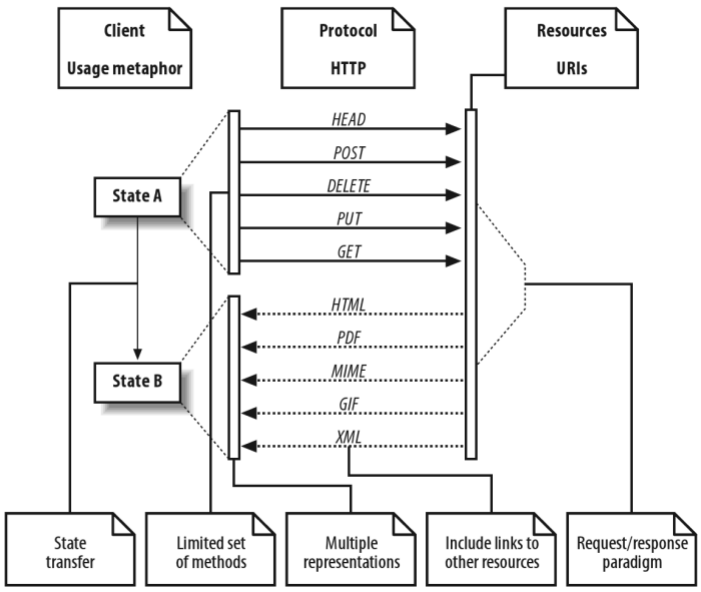


Figure 2.1: REST Example Architecture. Source: [1]

For example, in the case of a request to an URL on the web*:*
GET is used to receive the current repesentational state of a resource from the server*;*
PUT is used to transfer the current repesentational state of a resource to the server;
POST is used to create a new repesentational state in the server;
DELETE is used to delete a new repesentational state from a server;

For these reason, REST is a popular lightweight alternative to mechanism like RPC (Remote Procedure Call) and SOAP, particulary in M2M area where more and more architecture designers have adopted RESTful architecture [22, Section 7].

2.2.2 Publish/Subscribe

Publish/Subscribe is a communication model where data is delivered from publisher (data/ event producers) to subscribers (data/event consumers) in a decoupled way. In fact senders (publish-

ers) cannot deliver a message to a receiver (subscriber) directly, but use a third entity called *Dispatcher* or *Broker*. The broker is responsible for forwarding messages from senders to the registered receivers following some criteria.
Publisher introduce data events into the system performing a *Publish* operation. The generated message, that specifies values for a set of attributes associated with the generated event, is sent to the broker specifying a particular subject or topic where it must be dispatched. The publisher is also completely unaware of the subscribers that will receive this messages.
To be able to receive messages, receivers instead must perform a *Subscribe* operation to the Broker indicating which topic they are interested in. This require that a receiver knows available topic in advance.

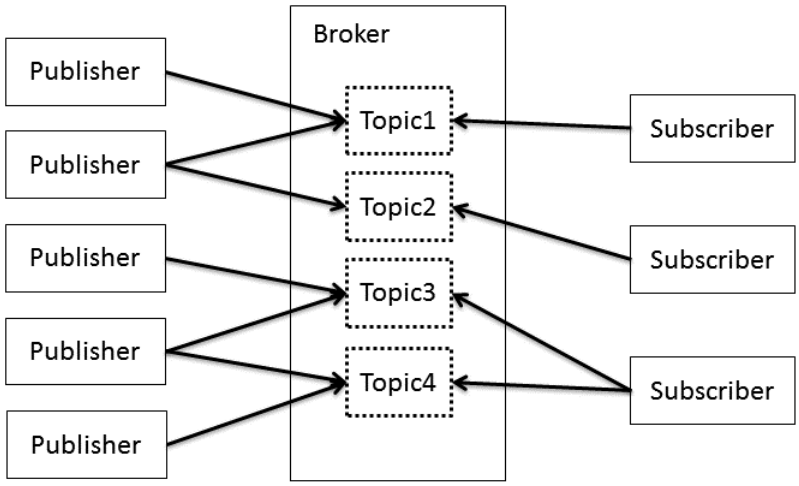
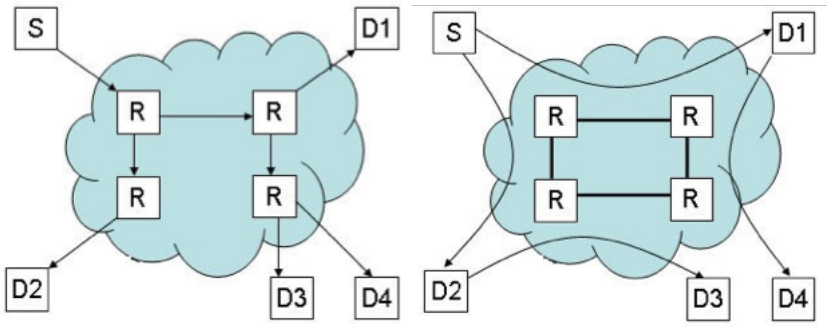


Figure 2.2: Publish Subscribe Model

In topic-based system, once the broker receives a message from a publisher for a given topic, it fetches the list of subscriber for that particular topic and forward the message to them.
In content-based systems instead, subscribers specify attribute of the data events as filters to be used by the broker.

Since subscriber can register to one or more topic (or event attributes/properties) and receive only the messages they are interested, this mechanism allow to provide great scalability to the system/network.
For this reason publish/subscribe systems are used in many applications including online stock quotes, Internet games, and sensor networks.

A relevant example of Publish/Subscribe protocol is Message Queue Telemetry Transport (MQTT). MQTT minimizes network bandwith and device resource requirements while attempt-



(a) IP Multicast (b) ALM Multicast

Figure 2.3: Multicast Approaches

ing to ensure reliability and delivery. This approach makes the MQTT protocol particularly well-suited for connecting M2M and IoT devices.

2.2.3 Multicast

Multicast is a group communication method in which one node of a network transmits information to several other nodes belonging to a group, using a single transmission. As shown in [4] there are different way to implement a multicast communication:

- IP multicast: when joining or leaving a group, the host informs its designated multicast router, then the participating multicast routers exchange between themself information on group membership and process it in order to establish multicast trees for data delivery (Figure 2.3a);
- Application Layer Multicast (ALM): basically an implementation of multicasting functionality as an application service instead of a network service. The multicast architecture, the group membership and data forwarding are controlled only by the participating end host as shown in Figure 2.3b;
- Overlay Multicast (OM): it is a solution halfway between IP Multicast and ALM. Service nodes establish multicast trees in order to provide Multicast functionalities. Proxies not in the backbone overlay can deliver packets via multicast or unicast;

IP Multicasting has great potential but, it has some deployment issues, shown in [23], that prevented it from becoming available on a global level. For example, IP Multicast routers need to be available at all levels of the network for the multicasting service to work but, in order to

handle high level of traffic, there is now the inclination to install very simple and fast routers at backbone level. There also exists management and security issues related to IP Multicast: flooding, unauthorized reception of data, avoiding the same multicast address for two sessions, etc.

Metrics	IP	ALM	OM
Ease of deployment	Low	High	medium
Multicast efficiency	High	Low	medium
Control overhead	Low	High	medium

Table 2.1: Comparison of Multicast Architectures (Source:[4])

Regarding ALM, longer delays and less efficiency in network usage of IP multicasting are minor disadvantages compared to advantages such as immediate deployability on the Internet, easier maintenance, update of the algorithm/software and the ability to adapt to a specific application. The Overlay Multicast has the same deployment issues of IP multicast and part of the advantages of ALM.

2.3 CoAP

CoAP is a application layer transfer protocol defined by IETF's *Constrained RESTful Environment* (CoRE) working group [2]. CoAP is based on a REST architecture [24] in which the server resources are identified by Universal Resource Identifiers (URIs) (see Section 2.2.1). The resources can be manipulated using the same methods provided by HTTP: GET, PUT, POST and DELETE.

CoAP was developed as communication protocol optimized for resource constrained networks; it uses a subset of HTTP functionalities that have been re-designed for small embedded devices. It has also been added other functionalities useful for IoT and M2M environment.
An example of such optimization is the header; it requires only 3 bytes for definig basic information like *Message Type* and *Request method/Response Code*, needed to provide asynchronous message exchanges, and *Message ID*, used to avoid duplication (Figure 2.4). Another byte is used for the *Token* to match response and request and after it *Options*, similar to HTTP Headers and used to transport metdata, may be listed. Lastly the payload (if any) follows.
A small header allows for a small message overhead and lower complexity, thus making it more efficient in terms of time and energy.

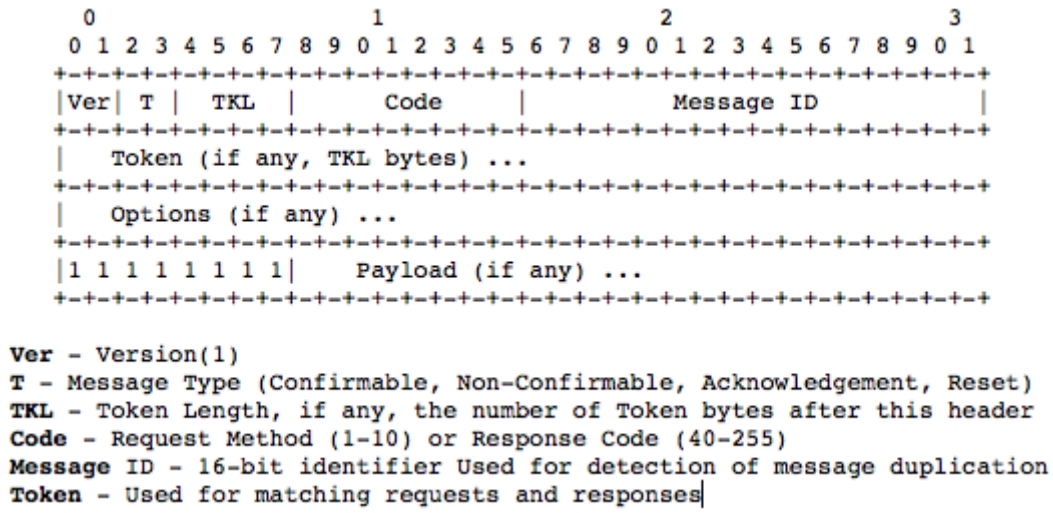


Figure 2.4: CoAP Header (Source: [2])

CoAP defines four message types for request and response, which are sent over UDP. For requests we have a *Confirmable* message (CON) and a *Non-confirmable* message (NON); the first type requires an acknowledgment response, the second does not require any response. For responses CoAP defines the *Acknowledgment* (ACK), used to acknowledge a CON message and a *Reset* message (RST) to indicate that a CON message has been received but could not be processed.

CoAP also provides a Resource Discovery feature, making discoverable which resources are offered by a CoAP endpoint. The basic method provided by the protocol is to use the URI-prefix "/.well-know/" but other ways are currently being defined[25]. The standard also states that "[in order] to maximize interoperability, a CoAP endpoint SHOULD support the CoRE Link Format of discoverable resources" as defined in [26], therefore the Resource Directory, to which endpoints are assumed to proactively register and maintain resource directory entries, is the main resource discovery mechanism, compulsory in all CoAP endpoints.

Lastly another important feature provided by CoAP is the Observation mechanism [27]. This functionality allows to mark a resource on a server as *Observed* and then constantly receive updates if the observed resource changes value (Figure 2.5). These updates are in the form of a new response from the server to the Observe request, by using the same *Token*. In this way clients are able to obtain the latest value of a resource.
This mechanism, which implements a publish/subscribe pattern (see section2.2.2), is obtained by adding the Observe option to the (first) GET request and then to all the subsequent, if any, responses (*Notification*) which will also use the same token of the request.

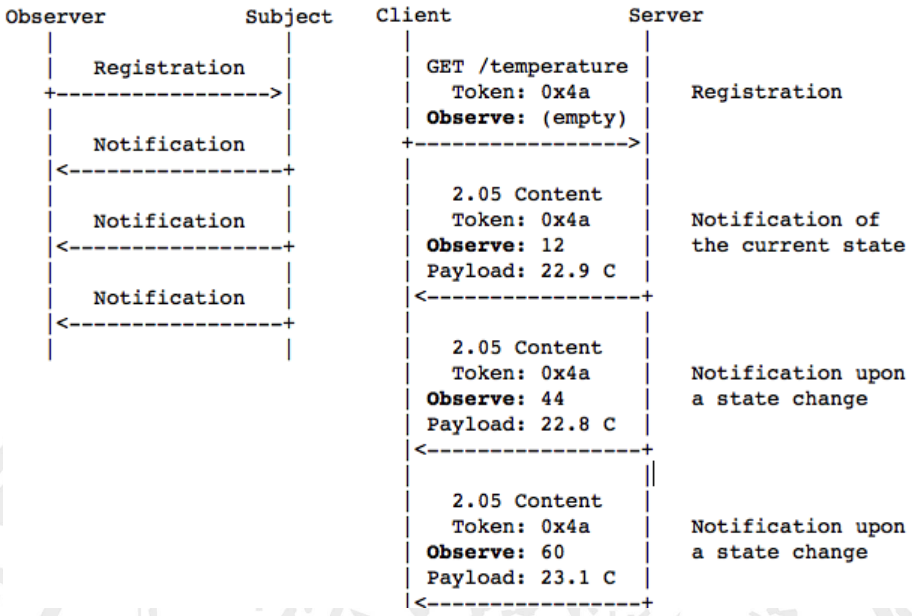


Figure 2.5: CoAP Observe Example (Source: [2])

2.4 Management Protocols

The moment any computer network becomes larger than a single LAN and a few PCs, a Network Management System (NMS) is required. A Network Management System is a set of equipment and/or application, based on standardized network-management protocols, that allow to monitor and manage single elements of a network [28].

Usually, running on managed devices there is a software that act as interface between the NMS and device itself: the *agent*. An agent provides management information to the NMS by keeping track of various aspects of the device. A NMS is responsible for querying agents in the network and also of receiving events from them if something of interest happens. These informations can then be used to keep the network (and the services that the network provides) up and running smoothly and also to take appropriate action is something bad occours.
Below, an introuction is giver for the two de-facto standard for network and device management, TR-069 and SNMP, and the new Lightweight OMA DM.

2.4.1 SNMP

Simple Network Management Protocol (SNMP) is an application–layer protocol for network management defined by the IAB (Internet Architecture Board) in [29]. SNMP, using UDP as transport layer, exchanges management information between network components in order to control devices for network communications, like routers, switches, servers, modem and many

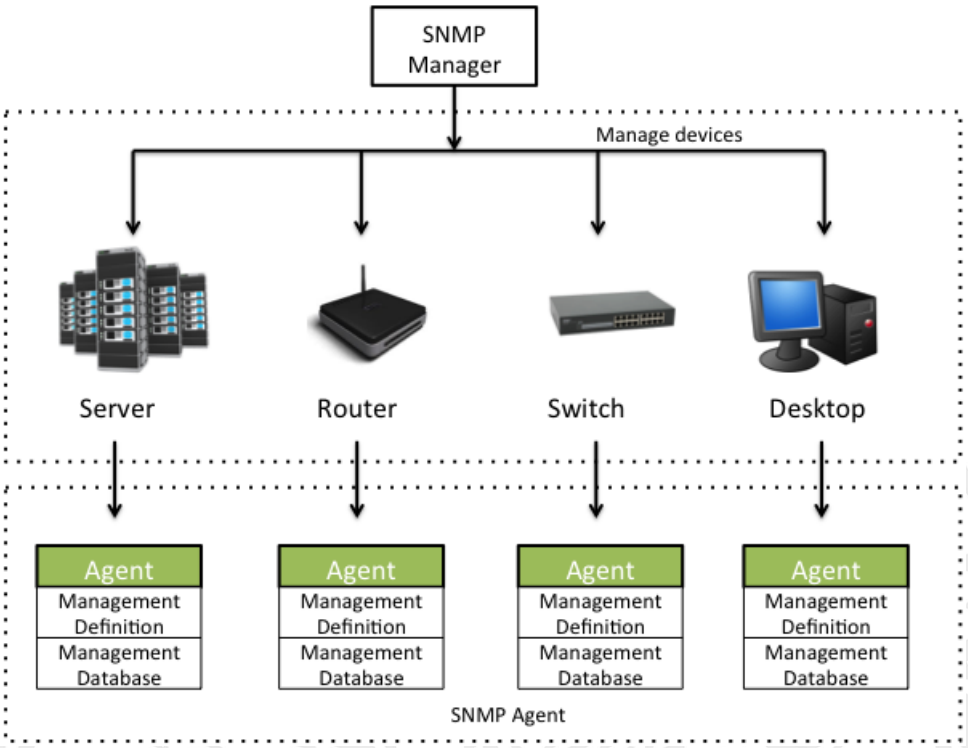


Figure 2.6: SNMP Components

other devices [30]. SNMP is now one of the most widely accepted protocols to manage and monitor elements distributed throughout a network.

2.4.1.1 SNMP Entities

SNMP has four principal components (Figure 2.6): SNMP Managers, SNMP agents, Managed devices, Management Information Bases (MIB).

SNMP Manager: The SNMP manager communicates with the SNMP agent with SNMP messages, and is responsible for retrieving information (for monitoring) and request modifications of the configuration of managed devices.

SNMP Agent: The agent is an application running on managed devices that responds to information request and performs actions requested by the Manager. In order to satisfy these request the Agent has to collect management information in a structured database, the Management Information Base (MIB). It is also possible for an agent to send events to the manager or act as a proxy for other network nodes which do not support SNMP.

Managed device: The managed device is the network element running the agent the SNMP Agent and wich the SNMP Manager manages.

Management Information Base (MIB)[31]: The MIB is a database describing the internal state of the managed device and the interface used by the manager. The MIB is composed of managed objects identified by an OID (Object Identifier) and hierarchically structured (Figure 2.7). Each OID identifies a specific property of the managed device that can be configured by SNMP. Since the MIB is organized hierarchically, it can be represented in a tree structure with where every node is an OID, thus every Object is usually addressed using a dotted list of integers (i.e. UDP Object is '1.3.6.1.2.1.7').

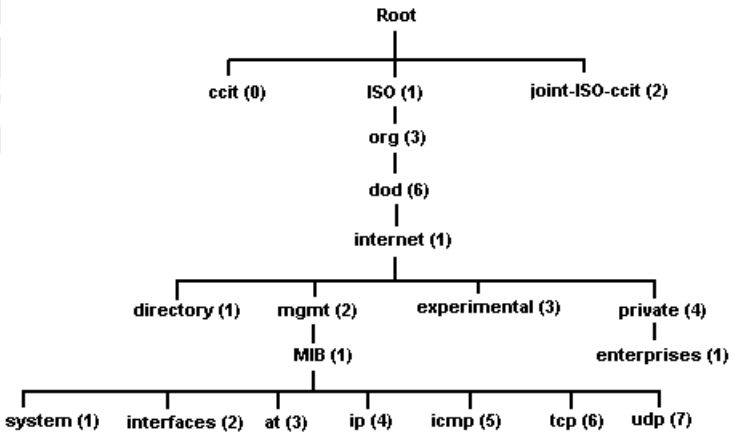


Figure 2.7: MIB example

2.4.1.2 SNMP Messages

SNMP defines a seven simple messages for communication between Managers and Agents: *Get, Get Next, Get Bulk, Set, Response, Trap, Inform.*
The Manager retrieves and sends information and configuration parameters using *Get* and *Set* messages on Agents. Variations of *Get* message are *Get Next* and *Get Bulk*: the first is used to get the value of the next OID; the second is used to retrieve a large amount of data in a single request instead of using multiple Get/GetNext request.
Response is the message used by the Agent to send back the value requested, or to report any erroneous information. When the Agent has to let the Manager know of some event, a *Trap* message is used to send a notification message. If an acknowledgment is required for the notification an *Inform* message is used.

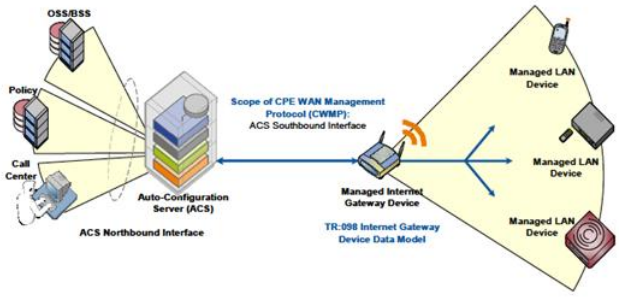


Figure 2.8: TR-069 architecture

2.4.2 TR069

The TR-069 CPE WAN Management Protocol (CWMP) [32] is a text based protocol created by the Broadband Forum [33] in order to standardize the Wide Area Network (WAN) management of CPE (Customer Premise Equipment).

The protocol define the communication between the CPE and an Auto Configuration Server (ACS) as shown in Figure 2.8. The latest is used for auto-configuration of the CPE, status monitoring, diagnostic of connectivity or service issue. Messages exchanged between the devices (CPE) and the ACS are transported over HTTP (or HTTPS). In this case the ACS act as passive entity, thus it is the CPE that must initiate the session and act as an HTTP client, while the ACS is the HTTP server (Figure 2.10).

Method name	CPE requirement	Server requirement
CPE methods	Responding	Calling
GetRPCMethods	Required	Optional
SetParameterValues	Required	Required
GetParameterValues	Required	Required
GetParameterNames	Required	Required
SetParameterAttributes	Required	Optional
GetParameterAttributes	Required	Optional
AddObject	Required	Optional
DeleteObject	Required	Optional
Reboot	Required	Optional
Download	Required	Required
Upload	Optional	Optional
FactoryReset	Optional	Optional
GetQueuedTransfers	Optional	Optional
ScheduleInform	Optional	Optional
SetVouchers	Optional	Optional
GetOptions	Optional	Optional
Server methods	Calling	Responding
GetRPCMethods	Optional	Required
Inform	Required	Required
TransferComplete	Required	Required
RequestDownload	Optional	Optional
Kicked	Optional	Optional

Figure 2.9: TR-069 RPC

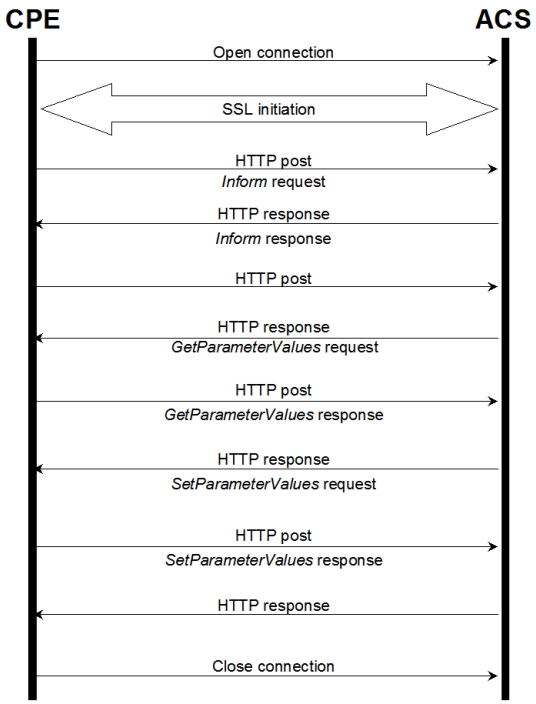


Figure 2.10: TR-069 message exchange

2.4.2.1 Session provisioning

TR-069 defines a list of Remote Procedure Calls (RPC) that the CPE must support in order to be able to communicate with the ACS. All the RPC are exchanged inside the provisioning session. The CPE is always the one to start the session with the transmission of a *Inform* message.When the ACS send a the same message as response, the orders can be transmitted from the ACS to the CPE or viceversa. The list of RPC available is shown in figure 2.9. When one entity has terminated its turn, which means it has no more orders to send to the other entity, it must send an empty HTTP message (request for the CPE, Response for the ACS). When this happens for both the entities, the provision session is considered terminated.

2.4.2.2 Data model

The data model used by TR-069 consist of an hierarchical structure that defines the parameter that RPC can set and get. Each level of this structure has its objects (or more object instances) and parameters. Every parameter is marked as writable or non-writable in order to know which one can be changed and which not. The key to access a parameter is constructed by concatenating with a '.'(dot) the names of objects (i.e. InternetGatewayDevice.WANDevice.{i}. WANConnectionDevice.{i}). The specification of this architecture is defined by the Broadband Forum in XML documents which defines type, value and meaning of every parameter.

2.4.3 LWM2M

Lightweight M2M (OMA LWM2M)[3] is a new slim client-server protocol with a RESTful data model. This standard was developed by the Open Mobile Alliance (OMA) in order to provide an application layer standard for device management and service enablement, which can be used in almost any use case, thereby across various industry segments.
LWM2M has been designed not only with the intent to manage low-cost constrained devices; the ones that run for years on one single battery charge, but also more powerful and expensive computing devices sych as smartphone or home routers. For both cases, the focus was on how M2M devices and their applications can be remotely managed.

2.4.3.1 Resource Model

The LWM2M protocol defines a data model (Figure 2.11) where information is represented by a *Resource*. A Resource is an atomic piece of information which can have multiple instances and that can be Read, Written or Executed. Multiple resources are logically grouped in an *Object*. A LWM2M Client has one or more *Object Instances*.

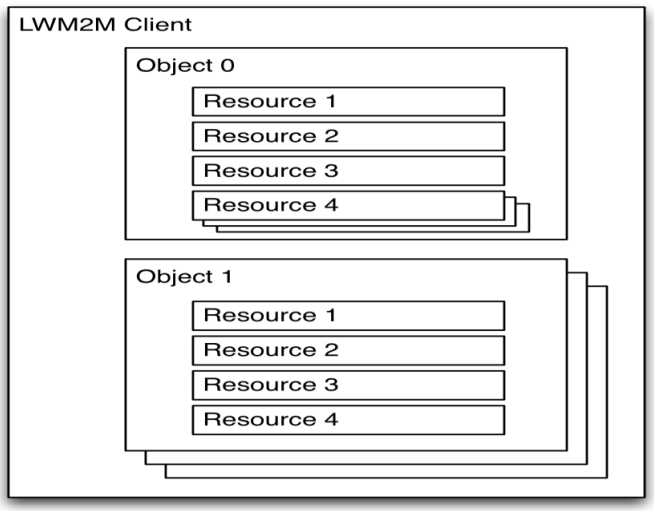


Figure 2.11: Resource Model

The access to the Client resources by the LWM2M Server is regulated by the Access Control List (ACL) Object which stores the permission of a server to access any resource in the LWM2M Client (figure 2.12). Objects, Object Instances and Resources are identified by a numeric ID and can be accessed with a simple URI in the format "/ObjID/InstanceID/ResourceID". For exempale "/1024/10/1" is an URI to access the Object 1024, Instance 10 an Resource 1.

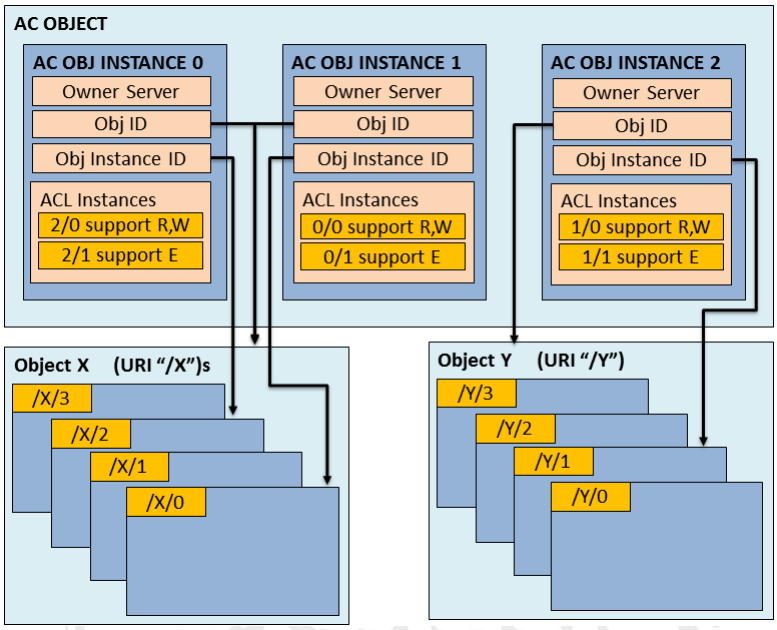


Figure 2.12: LWM2M Access Control

The version 1.0 of the OMA LWM2M standard specifies also an initial set of objects for device management purposes:

LWM2M Security: provides keying material of a LWM2M Client to access the approrpiate LWM2M Server.

LWM2M Server: provides the data related to a LWM2M server.

Access Control: is used to check whether the LWM2M Server has access right for performing an operation.

Device: provides some information on the device that can be queried by the LWM2M Server, as well as device reboot and factory reset function.

Connectivity Monitoring: enables monitoring of parameters related to network connectivity.

Firmware: includes methods needed in order to install and update firmware packages, and perform actions after updating firmware.

Location: the GPS location of the device.

Connection Statistics: enables the client to collect statistical information and enables the LWM2M Server to retrieve these information; also sets the collection duration and reset the statistical parameters.

One of the biggest benefit of LWM2M is the mapping between the CoAP RESTful commands (GET, PUT, POST, DELETE) and the LWM2M data model. In this way is possible to extend the data model for any kind of M2M use cases specified by CoAP. New compatible objects can be created to provide new use case for M2M devices and appliances. For example the IPSO Alliance [34] has already created object descriptions related to smart city applications [35]. The OMA has lanunched a public website [36] where these Objects are available and can be registered free of charge. Registered object will be listed in the OMNA LWM2M Object and Resource Registry.

2.4.3.2 Architecture

LWM2M is an application layer protocol that defines the rules of communication between a LWM2M Server and a LWM2M Client. The LWM2M Server is meant to be hosted on one or more machines of the M2M Service Provider (Figure 2.13). The LWM2M Client is meant to be hosted on a device as module or library.

LWM2M utilizes the IETF Constrained Application Protocol (CoAP) [2] as the underlying transfer protocol, over UDP and SMS. As illustrated in Section 2.3, CoAP defines message header, options, request/response codes and retransmission mechanisms. LWM2M uses a subset of response codes of CoAP and they are used to identify response messages.
Built-in resource discovery is supported using the CoRE Link Format standard [26]. The CoAP header is encoded in binary, so all functionalities are provided with minimum overhead. As transport layer to be used below CoAP, LWM2M defines UDP as mandatory while the SMS transport is optional. This means, that LWM2M messages can be sent both via SMS and UDP.

Regarding security, Datagram Transport Layer Security (DTLS) [37] is used to provide a secure channel for the messages exchanged between the Server and Client. DTLS security modes include both pre-shared key, Raw Public Key or X.509 Certificates [38] supporting simple and more complex devices.

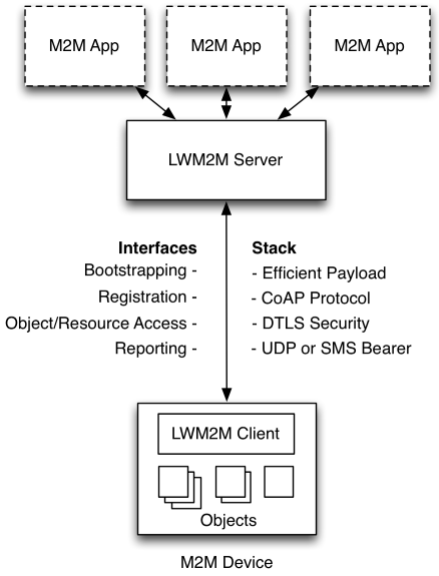


Figure 2.13: LWM2M Architecture (Source: [3])

2.4.3.3 Interfaces

The LWM2M standard defines four logical interfaces (figure 2.14):

Bootstrap: allows a LWM2M Bootstrap Server to provide the Device (LWM2M Client) with information for estabilishing a secure connection to the LWM2M Servers, like Server URI, encryption key and Access Control List. These information could also be retrieved from the Device flash memory or from a Smartcard.

Registration: Used by the LWM2M Client to register to one or multiple LWM2M Server, to provide information about how to contact the LWM2M Client (Binding Mode) and to learn which Object are supported from the LWM2M Client side.

Device Management: used by the LWM2M Servers to access Object Instances and Resources available in the LWM2M Client. The operations defined are "CREATE", "READ", "WRITE", "EXECUTE", "WRITE ATTRIBUTE", "DISCOVER", "DELETE". The operations supported by a particular Resource are definend in the Object Definition using the Object Template available in [3, Appendix D.1].

Information Reporting: allow the LWM2M Server to *Observe* [2] a Resource or Object Instance in the LWM2M Client. When a change occours the new value is sent to LWM2M Server in a *Notify* message. An observation ends when a *Cancel* message is sent and executed.

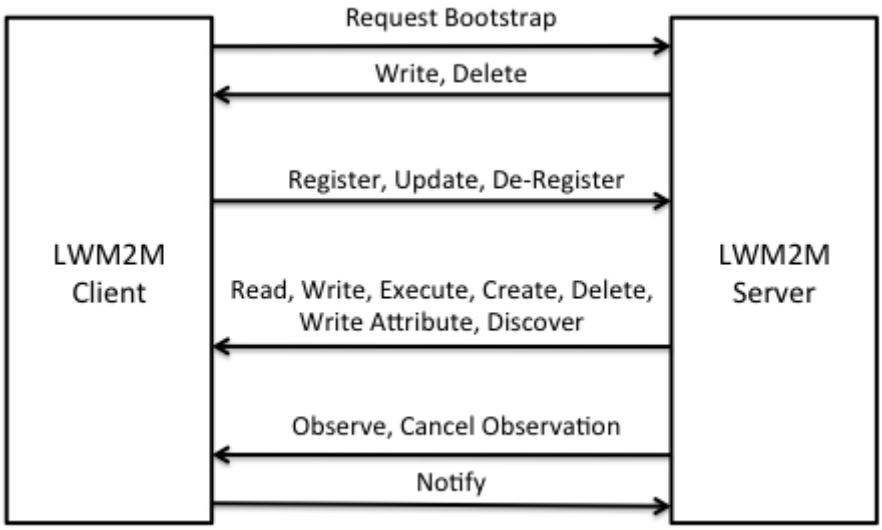


Figure 2.14: LWM2M Interfaces

Chapter 3

Design

The goal of this thesis is to show how LWM2M could be extended and improved in order to adapt the protocol for larger numbers of managed devices. It has been found necessary during the thesis research to focus on aspect such as group management, aggregation and network optimization. We assume a scenario as shown in 3.1 in which we have multiple M2M devices, which will support the LWM2M protocol by implementing a LWM2M Client. The manager of the M2M devices is located on the Internet and runs at least one LWM2M Server. The manager and the M2M devices are connected by gateways which are transparent in the LWM2M layer.

Under this assumption, a LWM2M Proxy is introduced in the gateways. This LWM2M Proxy is a node which has both LWM2M Client and Server capabilities, and it is responsible for forwarding the commands sent by the LWM2M Server as well as aggregating the subsequent responses from LWM2M Clients. The Proxy node also provides functionality for group management and information reporting and provides information to the manager about the connected M2M devices. All these functionalists (forwarding, aggregation, group management etc.) are encapsulated in Object which are accessible by the Client-side of the Proxy.

3.1 Motivation

LWM2M is a protocol for managing constraint devices, and it is designed to be a one-to-one communication protocol between a LWM2M Client and a LWM2M Server, not many-to-many. Extending the LWM2M protocol is necessary for the following reasons:

- It is not possible to aggregate request nor responses. M2M devices are supposed to be resource-constraint, particularly regarding battery. A middle entity between the client

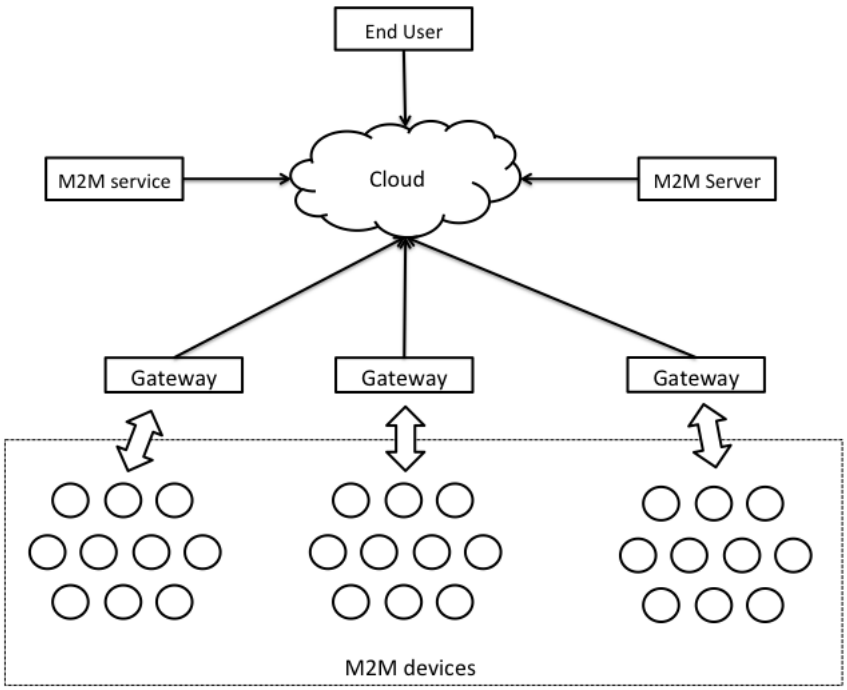


Figure 3.1: M2M Architecture

and the server would allow the use of aggregation mechanisms for LWM2M requests to a single client from multiple servers (using queue mode of LWM2M for example) and caching techniques for responses (for example of information reporting messages) from devices.

- The protocol does not provide a direct way for managing group of devices. Constrained devices can be large in numbers, but are often related to each other in function or by location. For example, we can have different group of light switches based on the room where they are deployed and have another group type for all the temperature sensors in the house. Groups may be preconfigured or dynamically set up during operation. If information needs to be sent to (or received from) a group of devices, the use of grouping techniques can reduce latency and bandwidth requirements.

- Introducing a LWM2M Proxy into the system simplifies design of the management logic both in the Client and the Server. Since proxy nodes will be in general more capable in processing power and with fewer battery constraints than the M2M devices on which the LWM2M client is running, it is straightforward to shift the complexity to the proxy.

3.2 Objective and Requirements

Group communication involves a one-to-many relationship between LWM2M endpoints. Specifically for our purposes, a single LWM2M Server should simultaneously Read, Write or Execute resources from multiple LWM2M Clients. An example would be reading the battery level of all devices in a room simultaneously with a single LWM2M group READ request, and handling all the subsequent responses. To obtain such result, requirements are:

1. Group discovery: it should be possible to discover groups (e.g. find a group to join or send a multicast message) or to discover members of a group (e.g. to address selected group members by unicast);
2. Retrieve information from group properties (e.g. related resource descriptions);
3. Create and remove a group (statically and dynamically): it should be possible to define new group when needed, manually by the end-user or dinamically by following some rules;
4. Add, remove and enumerate group members: in the same way of creation and removal of a group, it should also be possible to add and remove a member from/to the group and also enumerate all members of a group.
5. Operation mapping for groups: when a LWM2M operation is sent to a group, this must result in the operation being executed on all members of the group.
6. Optional Reliability: the application can select between unreliable group communication and reliable group communication.
7. Access control primitives: only authorized entities can manage groups and perform operation in/on them.
8. Robust group management: the group management functionality should account for failure or sleeping node situations.

3.3 Guidelines

When developing a software, or more in general a project, it is important to define which principles should be followed in the process. The design and implementation phases were carried out while following these guidelines:

1. Minimal specification and implementation overhead: a group communication solution should re-use existing/established protocols and (software) components that are already installed on constrained nodes or that do not require too much effort to be added to the system;

2. Interoperability: devices not supporting group communication solution should be able to operate normally in the system anyways;

3. Scalability: Changing the communication model from one-to-one to one-to-many introduces a scalability problem, the provided solution should be scalable and should operate well even when the number of nodes increases;

4. Security: Group communications shall provide the same or higher security level as unicast communication has;

5. Efficiency and Optimization: the solution should

   (a) Improve energy consumption of the constrained devices;

   (b) Limit the consumption of CPU or Memory;

   (c) Delivers messages more efficiently than a "serial unicast" solution;

3.4 Architecture

The current architecture of LWM2M for one-to-one communication is shown in Figure 2.13. Moving to a one-to-many communication requires that all messages destined to a group is multicasted to all the group members. In Section 2.2.3 we have seen that there are different way to implement a multicast communication: IP multicast, Application Layer Multicast (ALM), Overlay Multicast (OM), with different requirements, advantages and disadvantages.
We have seen that IP Multicasting has deployment requirements and it has management and security issues, instead ALM provides immediate deployability and it is easier to maintain and to update the algorithm/software, which is a desirable requirement in M2M and IoT systems. The Overlay Multicast has the same deployment issues of IP multicast, and part of the advantages of ALM.
We also know that CoAP protocol, which LWM2M uses as lower layer (see Section 2.4.3) identifies IP Multicast as main multicast solution[39] but does not exclude the other two alternatives [40] and we want the solution to be working independently from the lower layer. For such reason the ALM option is then chosen as multicast solution.
As shown in [41], there are several approaches for a ALM solution. The one chosen for our

implementation is a proxy-based solution, thus we introduce a new entity, the LWM2M Proxy, between the Server and the Client that will act as Group Manager, the proxy will manage subscriptions to the LWM2M groups and the messages directed to member of a group. It will also be responsible for queueing, prioritizing and aggregating messages to single nodes.

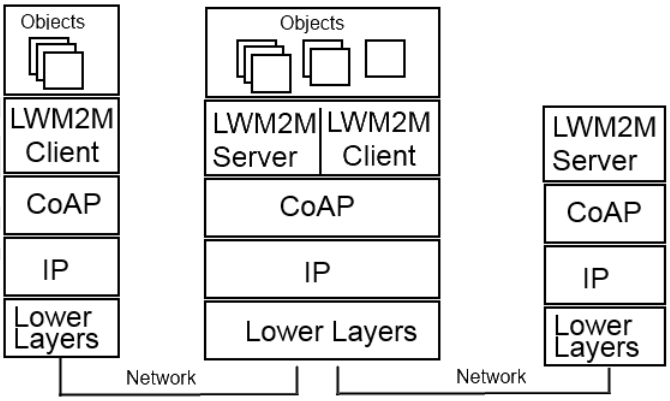


Figure 3.2: LWM2M Proxy

Introducing a LWM2M Proxy into the system simplifies the design of management logic in the Client and the Server. Since Proxy nodes are supposed to be more capable than the M2M devices (no energy and processing power constraints) on which the LWM2M client is implemented, it is straightforward to shift the complexity towards the proxy.

The proposed structure for the LWM2M Proxy is shown in Figure 3.2. The Proxy is an entity capable of communicating with both a LWM2M Client and Server and act as needed counterpart (Client or Server) when talinking with one or the other. Thus a LWM2M Proxy can be seen as the union of a Client and a Server in the same place, with the difference that the Server-side of the Proxy is able to access and modify the value of the Object and Resources defined in the Client part. This allow to implement the Interface of the Group Communication Service as one or more LWM2M Object of the Proxy. Further details will be provided in the next section.

For the Proxy to be able to act as Group Manager it is required that the Client nodes are registered to the Proxy (Server-side) and that the Proxy (Client-side) is registered to a LWM2M Server (which could also be a Proxy).
As shown in the registration hierarchy example in Figure 3.3, is possible to build hierarchies of Proxies, since there are no constraints on the number of Servers to which a client can register to (at the LWM2M layer, at least) and it also possible to have clients connected both to a Server and a Proxy.

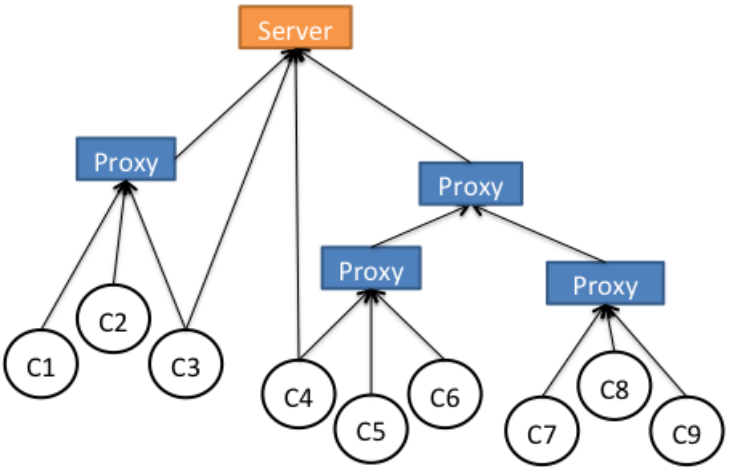


Figure 3.3: LWM2M Proxy

3.5 Group Mechanism

Having defined the requisites (Section 3.2), the architecture and since we have introduced the new entity "LWM2M Proxy", in this section details of the Group Communication Service are explained. First we will define a *Group* and how groups will be identified, then we will explain how the Group Request/Response mechanism works and lastly, details of the logic for the Group operations will be explained.

3.5.1 Group Definition

A LWM2M group is defined as a set of LWM2M nodes, where each node in the Group is able to receive LWM2M requests, thus both LWM2M Clients and Proxies can be members of a group. Being able to receive requests means that the node is registered to a Group Manager (Proxy) and it is not in an unrecoverable error situation. Sleep mode is a valid situation, infact when the device wakes up it should be able to process any queued message.

Each Group is identified by a number which will be unique for the M2M domain managed by the LWM2M Server. In CoAP all group communication operates through a group URI, that is a segment that must be added to the destination URI in order to identify the addressed group [39]. In LWM2M this solution is not viable since the LWM2M standard currently defines only the registration ("/rd") and bootstrap ("/bs") path, thus in order to allow our solution to comply with the current specification, instead of creating a new specific set of of commands for addressing groups, which would require standard modifications, we have chosen to match the definition of

group to a LWM2M Object.
This matching allows us to handle a group as an Object Instance in the LWM2M Proxy. This means that every operation defined for a Group must be translated as an operation to a LWM2M Object Instance and its Resources, moreover a list of resources must be defined for this Group Object.
Since the list of resources would be too long and chaotic we will split Resource based on the functionality provided. This translates in different Group Object based on the functionality they provide and we postpone the definition of their resource list to the next sections.

3.5.2 Group Management

After having defined a group in Section 3.5.1 and explained the mapping between group and object, we have to state how a Group is created, removed and discovered. This is pretty straightforward because, as said, a group is an Object Instance:

Create We can create a group with a simple *CREATE* request to the Group Object with the list of members that we want to become a member and any other information useful for the setup. The list can also be empty and members can be added later. The exact list of information will depends on the final Template of the Group Object.

Delete In the same way, to remove a Group is possible to send a *DELETE* command to the Object Instance representing the group.

Discovery The known group (alas Objects and Object Instances) can be discovered by the *UPDATE* and *Trigger Update mechanism* provided by the LWM2M protocol [3].

3.5.3 Group Messages

All LWM2M requests to a group (alias a *Group Object* instance) are sent like normal requests to the Proxy. When the Proxy receives a request to a Group Object, it performs the needed LWM2M operation for the specific request (like Create, Read, Observe etc...). Usually this implies the creation of a new request to message for all the members of the Group (more details later).

Once the Proxy has gathered all the responses from Group members (or timed out), it creates a *Group response message* which, based on the type of request, can be Simple or Detailed:

Simple Response The success or the failure of the Group Operation is represented by the response code of the CoAP message sent to the LWM2M Server/Manager. This response is used for operations that do not require the detailed list of response codes or success/failed operation from every Group Member.

Detailed Response All the responses from the Group members are aggregated in a JSON structure similar to the following one:

```
{ "u": "/1024/10/1", "op":"R","missing":0, "r": [
    { "ep": "urn:uuid:f81d4fae-7dec-11d0-a765-00a0cf691e6b",
      "c": 205, "v": "5"
    },
    { "ep": "urn:uuid:4faef81d-7dec-11d8-a765-00a0b9166bd5",
      "c": 503, "v": ""
    },
    { "ep": "urn:uuid:f8fae1d4-7dec-11dC-a765-00a0a91e1bf6",
      "c": 205, "v": "15"
    }
]}
```

The previous JSON has a "header" composed of four fields: URI (u) of the operation, a char representing the operation (op), the number of missing responses (missing) and the responses (r). The latest is the array of responses in which every response is identified by the triple Endpoint (ep), CoAP code (c) and a value (v) which it has a different meaning based on the CoAP response code. If there is no error (success status code 2xx) the value is the response for the request, if there are errors (4xx and 5xx errors) the value can be void or give details of the error. This schema allows to both identify the client that sent a response and the value or error returned.
A simple model of the message flow can be observed in Figure 3.4

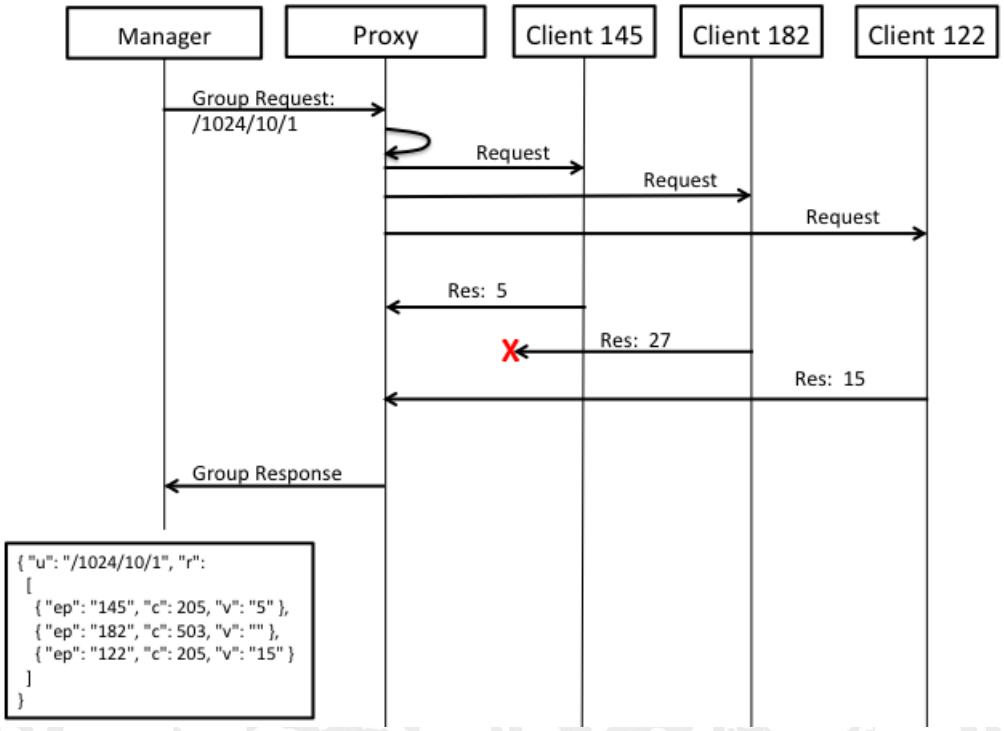


Figure 3.4: Group Request and Response

3.5.4 Group Membership

Once the Group is created we need to define how members can be added to a group, how they can be removed from it and how to provide member discovery. The best way to provide these functionality for our solution is to implement them as resources for the objects that will need them. A partial Object Template can be found in Table 3.1. Details are explained in later sections.

Name	Op	Instances	Mandatory	Type	Description
Members	R	Multiple	Yes	String	list of current members of the group.
Add Member	E	Single	Yes		Add one or more members to the group
Remove Member	E	Single	Yes		Remove one or more members from the group.

Table 3.1: Group Membership

Member Discovery

The main objective of Member Discovery is to know which clients (or Proxies) belong to a specific group. The simplest way to obtain this information is to maintain a list of members as a Resource of the Group. Since a group *is* an Object Instance stored in the Proxy, it is natural to think that it must also store the list of members as well. This is also very convinient because the Proxy also stores the registration and status information of registered clients. This way it is possible to avoid duplicated and inconsistent information.

The LWM2M protocol states that an URN (i.e. "urn:uuid:f81d4fae-7dec-11d0-a765-00a0cf691e6b") is used as identifier for an endpoint, formatted as defined in [3, section 6.2.1]. Being such URN unique at least in the M2M domain, we could not find a reason for not using such URN also for the identification of the Group member, since using another identifier would require a translation between identifiers which would prove counter productive for efficiency. For ease of reading, henceforth we will use simple numeric ID to identify an Endpoint instead of the longer UIID [42]. The template of the resource of the member list is shown in the first row of the table 3.1.

Membership Management

Regarding group membership management (adding and removal) of membes, since clients have only a passive role in the process, is possible to split the functionality in two separate cases: direct and indirect. In the first case the decision to add a or remove clients to/from a group is performed directly by the Manager (Server), in the second the action is performed by the Proxy, based on some criteria set by the Manager.

Direct Group Membership Management

In the case of a direct action from the Manager, we needed a way to let it add or remove a specific Client to or from a Group. Again, since the Group information as stored as an Object Instance of the Proxy and we can identify a Client using its URN, the most effective way for adding and removing members is to encapsulate these commands as two separate executable Resource in the Group Object following the template in Table 3.1. Doing so, we allow the manager to simply send an *EXECUTE* command to the correspondent resource with the list of URN belonging to clients to be added or removed as arguments. This solution is valid only for managing membership of a limited number of devices since it would be very inefficient to send multiple messages to add a list of clients.

Indirect Group Membership Management

Having the Server (Manager) to directly handle the membership of a group for a large number of devices is neither reasonable nor efficient. An improvement is to have the Proxy to automatically add or remove members to groups following some *policy*. A policy is a list of criteria that guide the decision process. In our case we chose to focus on criteria related to specific conditions or properties of managed devices, that is LWM2M Resources of the devices. Such decision is due to the fact that other criteria, not related to device properties, imply getting information from the internal state of the Proxy, which strongly depends on the implementation, or other elements of the proxy environment, which we cannot foresee.

Properties of managed devices and also LWM2M Resources are for instance the manufacturer name, the firmware version and the battery level; this set of information is defined in the mandatory *Device Object* (URI /3/0/0, /3/0/3 and /3/0/9 respectively) thus always defined in the devices. Also resources of optional Object can be used, i.e. latitude and longitude in the *Location Object* (URI /6/0/0 and /6/0/1 respectively). Using these information we can test their value and then group devices, for example, by Manufacturer, Battery level, Location, a combination or even all these criteria. The tests can be something like *less*, *greater* or *equal* for numeric value and a simple or partial *compare* for Strings. An example of Groups criteria is shown in Table 3.2.

	G1	G2	G3	G4	G4	G6	G7	G8	G9
Manufacturer	ARM	INTEL	INTEL				ARM	INTEL	ARM
Firmware			v1.5				v1.8		v1.1
Battery				>70	>20,<70	<20		>80	<40
Latitude		40.88			60.13				
Longitude		14.27			24.50				

Note: an empty cell means that the corresponding resource is not required

Table 3.2: Group Examples

Having defined which criteria will be used (the value of LWM2M Resources) and having stated that the process of membership management is performed by the Proxy on behalf of the manager, we have to decide how the Server/Manager will provide this criteria to the Proxy, when the process is started and what the process actually does.

Let us start with the process description. In order to start, at least one group must be defined in the proxy by the manager with a set of requirements that registered devices must met in order to be automatically added to the Group in consideration.
Let us assume we are executing the procedure of the Group G8 of the example in Table 3.2. In this case our requirements are {Manufacturer = "INTEL", battery level > 80%}. In this case the proxy has to request such information, that is to *READ* the corresponding URIs from the

devices unless they are already cached and still valid.
Requests can be made in series or parallel; the requests in series allow to stop the procedure if one condition is not met, thus to save some message. Requests in parallel instead allow to speed up the process. Once information is gathered and processed, if the device meets the requirements of the policies for the Group, it can be added to its members. An example of the communication flow is shown in Figure 3.5.

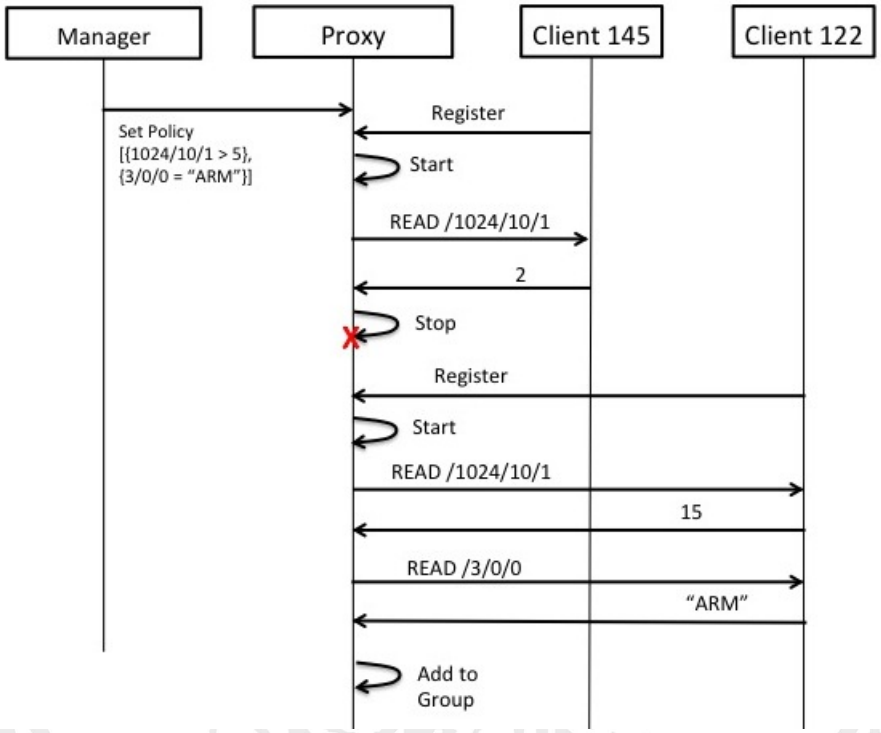


Figure 3.5: Policy check process

Once the policies are set, two possible outcomes are possibles: starting the process when the client registers or processing all registered clients from the last execution later on. Since the registration process is something that should not happen very often for each devices (or at least a very small percentage of times compared to any other operations) the additional load produced by the execution will be very small and the client will be added, if that it is the case, to a group as soon as possible. Instead if the process is performed after some time as a batch of operations, the load on the Proxy will be concentrated in the same period of time and the general logic will not have any benefit. For such reason the first approach was chosen.

About the *how,* to provide the criteria the best solution is (again) to have a Resource in a Group Object that will be used to store the Policies and that can be read and written. The format used for the policies is JSON and the structure chosen is similar to the following: {"u"="URI","o"="operation","v"="VALUE" }.

ID	Name	Op	Instances	Mandatory	Type	Description
1	Client List	R	Multiple	Yes	String	List of current members of the group
2	Policy	R/W	Single	Yes	String	Policies to be applied to the group
3	Update Trigger	E	Single	Yes		Update *Client List* using the criteria in *Policy*

Table 3.3: Group Membership Template

For example:

```
{"u"="/1024/10/1","o"=>,"v"="5" }
{"u"="/3/0/0","o" = "=","v"="INTEL" }
```

3.5.5 Message Multicasting

Having established how the Server manages group memberships, we now have to define the mechanism that enable the Manager/Server to send messages to the whole group. The messages we want to support are only the ones belonging to the LWM2M *Device Management* (DM) and *Information Reporting* (IR) interfaces (2.4.3.3), since to be able to use the Group Communication Service the device must have already completed the bootstrap registration procedure.
Since the communication model of the two interfaces is a bit different, the IR messages maps better to the Publish/Subscribe model than REST does (see 2.2.2), for this reason we divide the description of the mechanism in two parts. The first refers to the messages of the DM interface, the second the messages of the IR Interface.

Device Management (Forward)

Operations of the DM interface have all in common the fact that once the command is sent to a client and only one response (at most) is received; an IR *Observe* may produce more than one *Notify*. For this reason for the Device Management commands we can take advantage of the Group Request/Response Model described in Section 3.5.3, which is: the Server sends the command to be forwarded to the all the Group members to the Proxy; the Proxy creates a single request for all members; when all the members have sent their response or timed out, the Proxy aggregates the received responses and makes them available to the Server (i.e. sending them in an aggregated response).

Therefore what is missing is the interface of the Proxy. The simplest way to provide such interface is to add an executable resource to the Group Object template defined in Table 3.1.

Specifically, this resource, which we named *Forward*, should support only the Execute operation and, since it would counterproductive to encapsulate it in another CoAP message, then we used a serialized version of the following arguments: Command, URI and arguments of the command. In particular:

Command: The first letter of the command (thus one of the set {R,W,E,d,w,C,D}, with lowercased *w* is *Write Attributes* and lowercased *d* is *Discovery* to solve collisions)

URI the string version of the destination URI (i.e. "/1024/10/5")

Command argument the arguments that the client resource should accept, if defined. With all the arguments separated by a space.

Examples of serialized command are : "R /1024/40/5" and "W /4689/5/2 45".

When the message is received from the Proxy, it is then processed as described before. When responses from clients are received, the Proxy has to make them available to the Server, in a suitable format. We have two way to achieve this result:

1. Send the aggregated responses as a response to the Server request (piggybacked or separate CoAP response)
2. Save the aggregated responses somewhere and let the server fetch it when it need it (via READ command)

ID	Name	Op	Instances	Mandatory	Type	Description
0	GroupID	R	Single	Yes	Integer	ID of the Group
1	Members	R	Multiple	Yes	String	List of current members of the group
2	Add Member	E	Single	Yes		Add one or more members to the group
3	Remove Member	E	Single	Yes		Remove one or more members from the group
4	Forward	E	Single	Yes		Forward a command to the Group Members
5	Fw Response	R	Multiple	Yes	String	Aggregated responses of Forward requests

Table 3.4: Complete Group Object

None of the two solution has clear advantage over the other so, in other to make the response available to other server (useful in some use cases) we have chosen to save the aggregated response in a new resource (to be added to the Group Object template) and let the Servers decide if they want to receive the response as soon as it is available using the Observe/Notify mechanism. In fact if the Server is interested in receiving the response right away, before

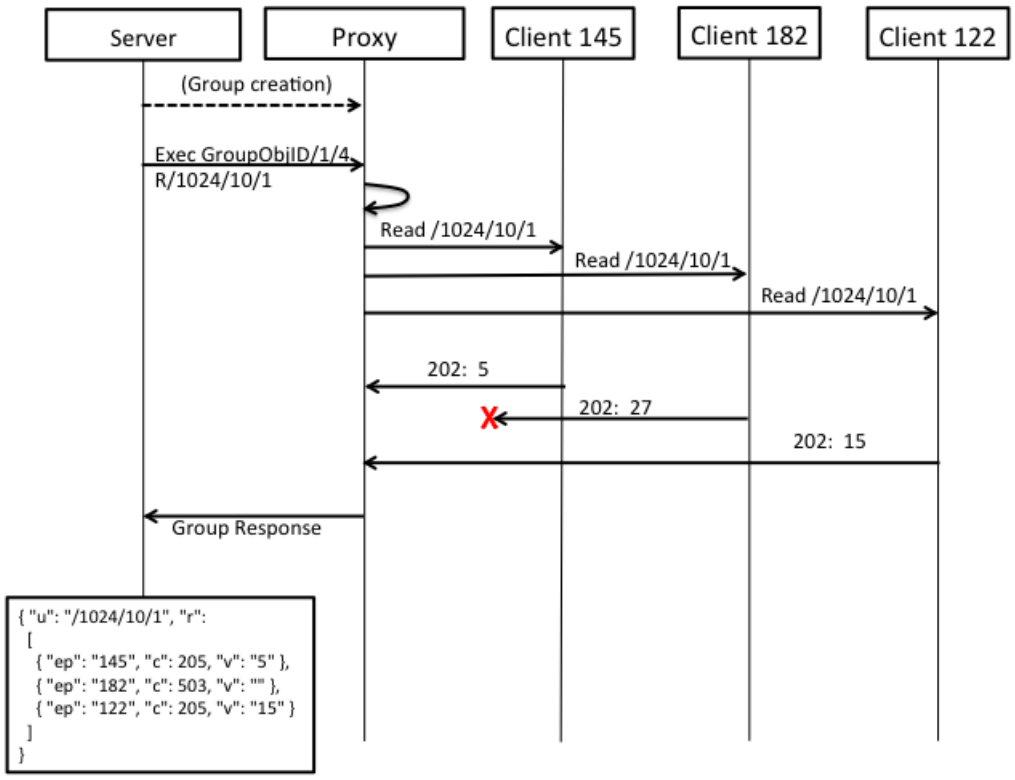


Figure 3.6: Group Forward message flow

sending the requests he can Observe the *Forward Response* Resource and receive a Notify message when an aggregated response of future group request is stored in it.

About the format of aggregated response, since the only information we need is, again, the tuple {client,code,value} we can use the same format defined in section 3.5.3. The complete Group Object is shown in Table 3.4, while an example of Group Forward execution is available in Figure 3.6.

Information Reporting (Observing)

As said before, the IR interface differs from the Device Management interface because once a Server is Observing a resource, it should receive more than one notification from the client. We will call the extension of the Observe operation to groups *Group Observe* and it consists in performing an *LWM2M Observe* on a resource of Clients belonging to group and setup the needed internal structures/processes to handle the notification and send them back to the Server. A Group Observe requires the Proxy to keep track of the transaction (observation) for a longer time compared to Device Management requests, which is until the first response/notify. That is

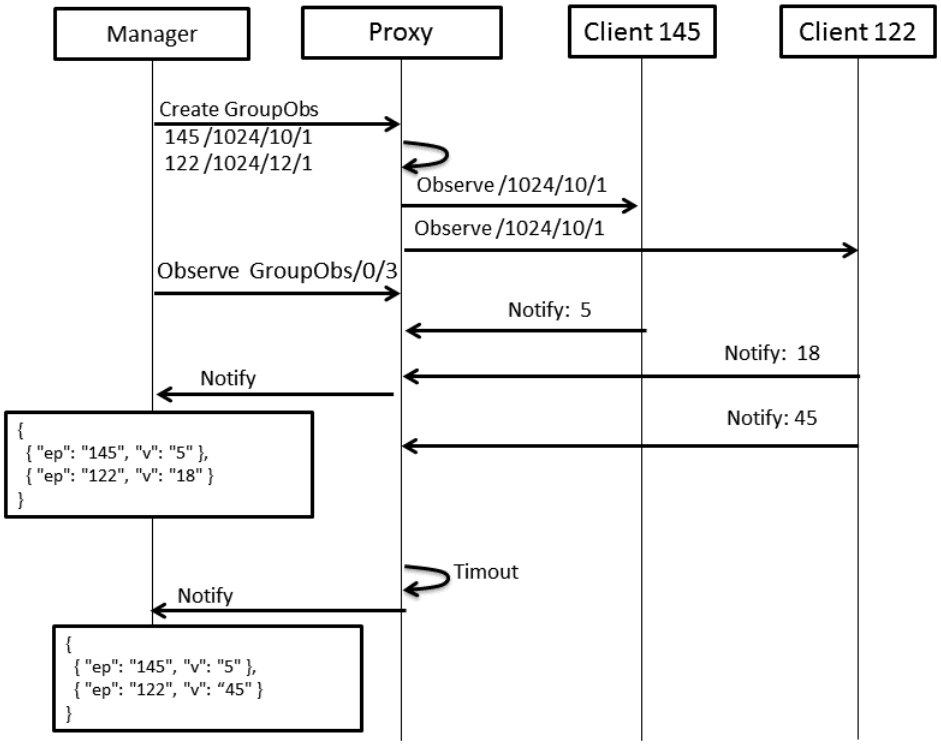


Figure 3.7: Group Observe message flow

the reason why in the design process we decided to shift the focus on the Group Observation and not the Group itself.

A Group request is an observation on multiple members, if we extend the Group Request/Response Model described in subsection 3.5.3, for the observe/notify case simply by considering multiple Notifies instead of a single response, we obtain the message model for the Group Observe (Figure 3.7).

Now we need to define the interface of the Proxy that let the Server perform the Group Observation. Since our focus is on Group Observe, we can encapsulate it in a LWM2M Object. In this way every Object Instance will match to a different Group Observation. To perform a new Group Observe request using this Object, the Server can just send a Create message (with the need parameter) to it, to stop the observation is sufficient to Delete the object instance representing the observation.

Before going further it is better if we define the Group Observe Object more accurately. The minimum information that we need for to perform a Group Observe is the list of Clients (ClientID) and the resources to observe (the URI). We can encapsulate the two pieces of information in

two separate resources in order to make the accessible. Moreover there is a connection between the *i-th* resource instance of one resource with the *i-th* of the other.

A Group Observation ID is used to identify group observations. In order to add and remove members (and their observed resources) two Execute resource can be added, a resource to store the value of the latest notifications for each Client observed can be used, were too there is a correlation between the *i-th* resource instances of Member and *Values* Resource.
A Recap of the resources need is shown in Table 3.5.

Name	Op	Instances	Mandatory	Type	Description
GroupObsID	R	Single	Yes	Integer	ID of the Observation
Members	R	Multiple	Yes	String	List of current members of the group
Resource List	R	Multiple	Yes	String	List of Observed Resource on every *Members*
Values	R	Multiple	Yes	String	Value Notified from *Members*
Add Member	E	Single	Yes		Add one or more members to the group.
Remove Member	E	Single	Yes		Remove one or more members from the group

Table 3.5: Group Observe Object

Having described the Group Observe, we have to define the arguments of the Create request. To make the request we need the ID of the client to observe and its relative resource to be observed thus the Create arguments will be a list of the pairs {ClientID,URI}. When the Proxy receives the Create request it will try to perform an observation on the provided list of clients and resources. If no error occur the object is created and a "201 - CREATED" response message is returned to the server, otherwise an error message is returned and the observation previously made are canceled.

After the Group Observation is executed, the server must perform an *Observe* request to the *Values* resource of the Group Observe Object. Doing so the Server will be able to receive a Notify message with the update value of the *Values* resource. This action also allows to correctly map the Group Observe to the Observe mechanism.
Clearly it is not useful to send a group notify whenever a new *Notify* is received from an observed client. It is more convenient to notify the server in one of the following cases:
1) when the notify of "all" the observed resources is received, or
2) when at least one Notify is received from the last update and a suitable period of time elapsed.
This "suitable time" can easily set by the server by taking advantage of LWM2M resource attributes (the *Values* resource in our case) which are meant to configure how often a *Notify* message must be sent to the server if a value change occurs.

Name	Op	Instances	Mandatory	Type	Description
Observed URI	R	Single	Yes	Integer	URI of the resource being observed
ObserverID	R	Multiple	Yes	String	List of current Observer of the Resource
MaxObserver	RW	Single	No	String	Set max number of observer for this resource

Table 3.6: ObservedMonitor Template

3.5.6 ObservedMonitor

Since the Group Observe Object introduces the opportunity to delegate an observation to another entity (the Proxy), it may arise the need of being able to monitor the observations that are in place on a Client. Providing a feature that satisfy this need would help avoiding redundant requests for observations from multiple Servers, since they may be able to exchange between themselves the information retrieved from clients.

We decided to provide this feature in the form of an object: the ObservedMonitor Object. Its main objective is to provide information on which entities are observing a resource. We defined three resources in this object: *observedURI*, *ObserversID* and *maxObserver*. The first two resources are enough for providing the needed functionality, in fact we have a match between an observation and a Object instance.
As extra feature we introduced the *maxObserver* resource in order to give a way to limit the number of observer for a specific Resource. In this way, it is possible to prevent some incorrect or malicious behavior that can drain devices battery or prevent the device to work correctly. The object template of the object is shown in Table3.6.

3.5.7 ProxiedDevices

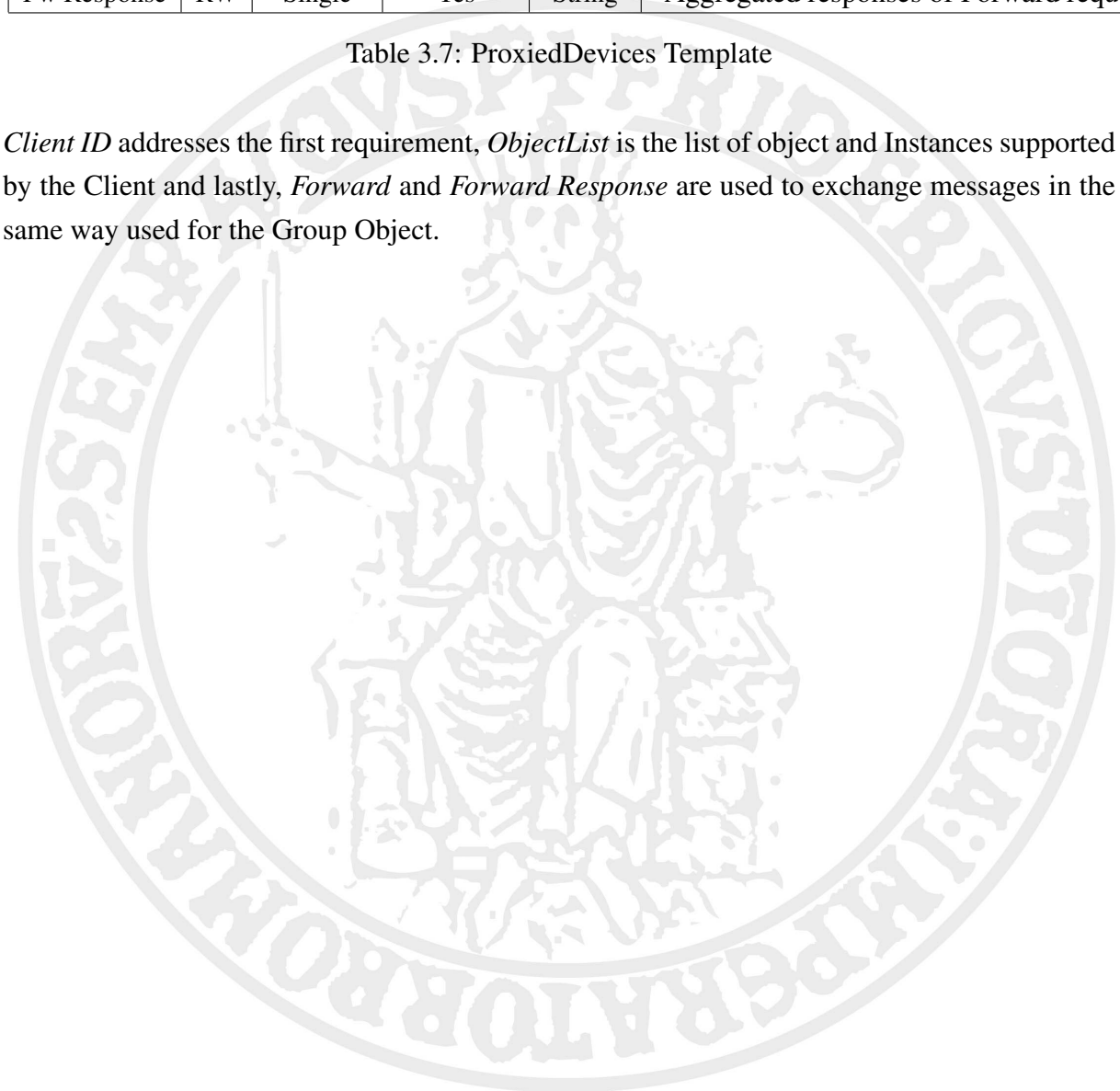
Another particular need that may arise in the system described in 3.3 is to send a direct message to a client through the Proxy, for example due to the missing direct connection between Server and Client or because the Client does not support LWM2M but it is somehow able to communicate with the Proxy (so the proxy is acting as interface). To cope with these conditions we defined a *ProxiedDevice* Object.

Requirements for this object are: to be able to address a specific client; to know which objects the client supports (which is equivalent to know which aspect of the client is manageable); and to be able to send messages and obtain a responses from it. This requirements have produced the template shown in Table 3.7.

Name	Op	Instances	Mandatory	Type	Description
ID	R	Single	Yes	Integer	Instance ID
ClientID	R	Single	Yes	String	Client ID or URN
Object List	R	Multiple	Yes	String	List of supported Object
Forward	E	Single	Yes	String	Forward a command to the Group Members
Fw Response	RW	Single	Yes	String	Aggregated responses of Forward requests

Table 3.7: ProxiedDevices Template

Client ID addresses the first requirement, *ObjectList* is the list of object and Instances supported by the Client and lastly, *Forward* and *Forward Response* are used to exchange messages in the same way used for the Group Object.



Chapter 4

Implementation

4.1 Introduction

In order to test the Group Communication Service, we have implemented a prototype for LWM2M Proxy, as well the new objects that were defined in the previous chapter. In this chapter we will first give a description of *Wakaama*, a LWM2M library used to implement our prototype, and the changes we made to it. Later we will describe how the Proxy and the corresponding objects were implemented and how the data structures were used in its internal mechanisms.

4.2 Wakaama (LibLWM2M)

Wakaama is an implementation in C of the Open Mobile Alliance's Lightweight M2M protocol (LWM2M). It started as an Intel Open Source project known as LibLWM2M [43], then it became an Eclipse Foundation project in June/July 2014 [44] and it is currently under development.

Wakaama it is not really a library, but it provides APIs through files to be build with an application, in order to provide support for the protocol. When we started the development of the prototype for this work, Wakaama was developed enough to provide basic functionality like: the Resource Data Model (missing ACL and property of resources), Registration Interface (*UPDATE* command missing), Device Management operations (*DISCOVER* and *WRITE ATTRIBUTES* missing), Information Reporting operations, TLV format for responses, transaction

and retransmission mechanism as well as a CoAP Library for building and parsing messages. A representation of the connection between these modules is shown in Figure 4.1.

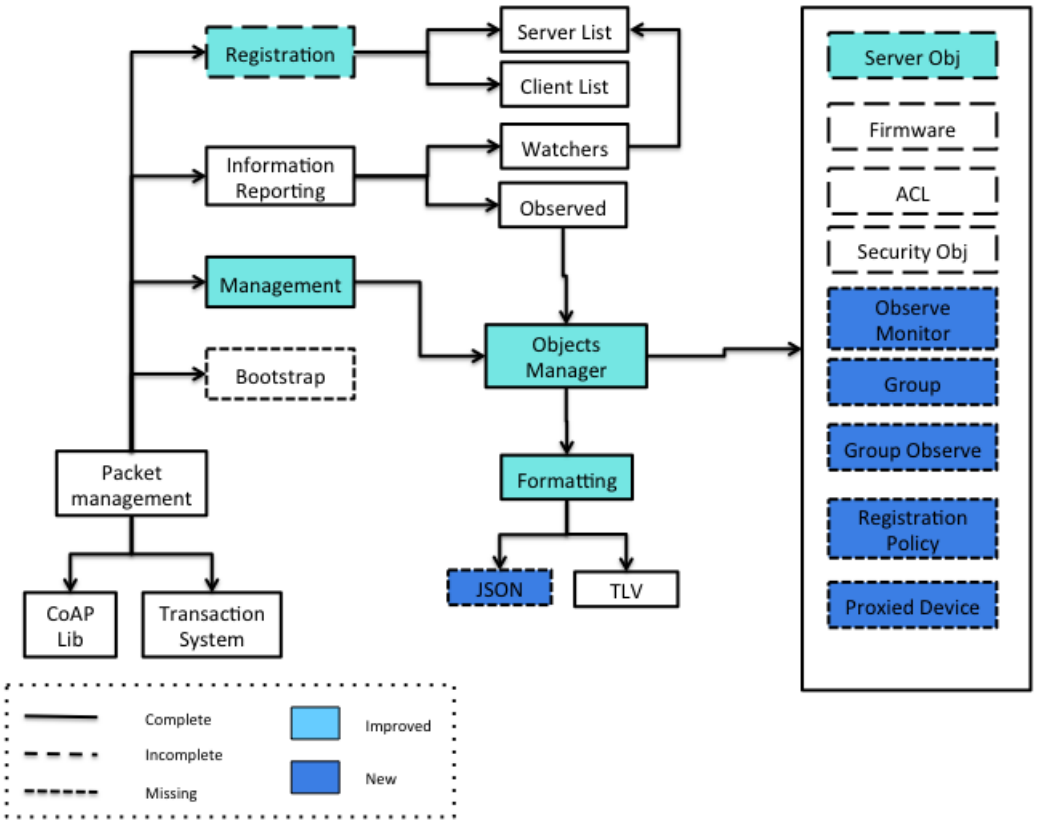


Figure 4.1: Wakaama Components

Wakaama uses conditional compilation (a compilation that produces different executables based on parameter provided during compilation) with *LWM2M_CLIENT_MODE* and *LWM2M_SERVER_MODE* constants in order to provide the application with the data structures and functions needed for the Client and Server interface respectively. All the information and data (ClientList, ServerList, defined Object and others) are accessible in the *lwm2m_context* variable which must be initialized to be able to use the library. After initialization, it is possible to define the connection from where packet will be received and, for the client, to add custom objects to the library or to add a list of servers to connect and register. After this, it is possible to process an incoming packet using the API *HandlePacket()* and use *lwm2m_step()* to perform any pending LWM2M operation.

Wakaama provides an example application for both client and server. They provide a simple terminal interface from where is possible to input commands. The command format is "*Operation* Client URI Data" for the server and only "*Operation* URI Data" for the client. In both cases this will invoke the correspondent LWM2M API with the provided arguments.

In the Server application, Wakaama takes care of building the CoAP packet corresponding to the selected command and URI and send it to the chosen (registered) Client. The server also store some of the request information in a transaction system in order to be able to process the response once it is received and also perform a retransmission.
For convenience, all registered Clients are stored in the Server and an *Internal ID* is assigned to them. In the Client application, Wakaama processes incoming CoAP packets in order to verify if the destination URI exists, it redirects the requests to the interested Objects and will perform the requested operation if it is valid for the resource and it is enabled to do so. Wakaama also builds and sends the response packet with the response payload or information errors if they occur. A simplified control flow is shown in Figure 4.2.

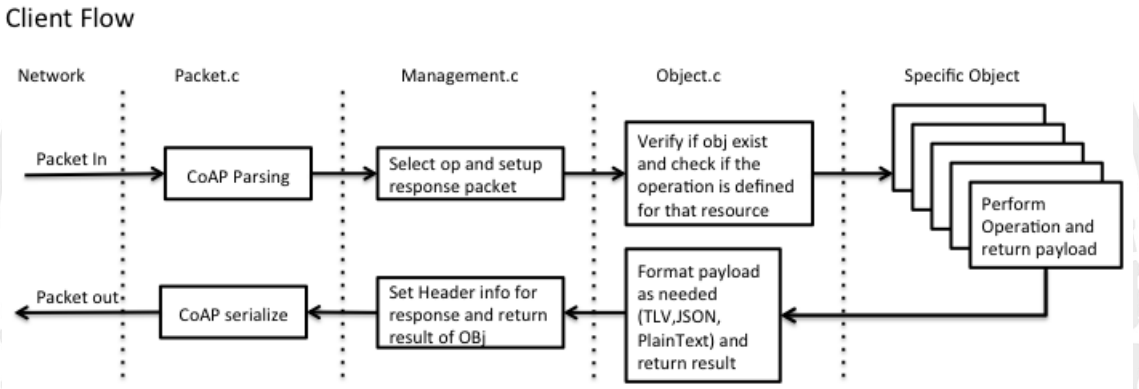


Figure 4.2: Wakaama Client

4.2.1 Improvement

Wakaama, being still under development, was missing some features that are necessary for our Group Communication Service Prototype. To solve this issue we modified the Wakaama (Core and Test application) and implemented the features that we needed, which are:

Create method In the server application we added the *CREATE* command to the command line interface of the test Server in order to be able to create Instances on the clients.

Update method The Update method allow the Client to send to the server updated information about object and object instances. We added this command to the library and test Client.

JSON The LWM2M standard states that the response format can be in TLV or JSON. Wakaama provided only the TLV format, we implemented also the JSON response format.

Server Obj We implemented a Server Object for the client, to be used in order to know to which server the client must connect and Register. Our implementation includes also the Update Trigger Mechanism ([3] Section 8.4).

Registration to multiple Servers having implemented the Server Object, we extended the test Client application in order to be able to connect and register to more than one server (we needed it for the registration to the Proxy)

4.2.2 Object Extension

As said before, Wakaama provides a set of APIs for implementing Client or Server applications. A subset of these API is used to implement new objects, in particular, to defin an object we must create a new variable of type *lwm2m_object_t*. This structure will store all data needed to process a request addressing the specific object. The structure and a brief description of its elements is shown in Algorithm 4.1.

Algorithm 4.1 Object and List structure

```
_lwm2m_object_t {
  uint16_t                objID;          /* Id of the Object*/
  lwm2m_list_t *          instanceList;   /* Head of the list of instances*/
  uint16_t                maxResId;       /* Max ID of Obj's resources */
  lwm2m_read_callback_t   readFunc;       /* Read callback*/
  lwm2m_write_callback_t  writeFunc;      /* Write callback*/
  lwm2m_execute_callback_t executeFunc;   /* Execute callback*/
  lwm2m_create_callback_t createFunc;     /* Create callback*/
  lwm2m_delete_callback_t deleteFunc;     /* Delete callback*/
  lwm2m_close_callback_t  closeFunc;      /* Close callback*/
  void *                  userData;       /* Store data for Obj inner Logic*/
};
_lwm2m_list_t {
  struct _lwm2m_list_t * next; /* pointer to the next node*/
  uint16_t     id;             /* id of the node*/
};
```

Object instances are organized as a lists. Wakaama does not define the structure for instances, which must be specified for every object, but lets the developer choose how to store information. The only requirement is that it should be compatible with the *lwm2m_list_t* type, in order to be able to perform a typecasting from the specific instance structure to the list type and viceversa, since Wakaaama internal mechanisms processes list ot this type.
Regarding methods callbacks, their argument are: the destination *URI*, the input/output *Buffer* (based on the type of command) and the object *userData.* All callbacks have as parameter the

specific object instance (based on the InstanceID of the *URI*) and they perform the requested operation on the instance data. The same schema was applied for new objects defined for the Group Communication Service.

4.3 Application Testbed

To be able to implement and test our Group Communication Service, thus test the Proxy and its new Objects, we need a Client and a Server application that provide basic features like supporting the LWM2M data model and the Registration, the Device Management and the Information Reporting Interfaces. Luckily Wakaama provides a test Client and Server Application that already have these features. Moreover, since CPU and memory requirements are very low (i.e. memory footprint below 300 KBytes), a simple low-end device is able to run hundreds or even thousand of Clients. That is why both a normal workstation and also a Raspberry Pi were used as testbed for our prototype. A brief description of the Client and Server logic follows.

Server

The test Server application provided by Wakaama has a command line interface for *Device Management* and *Information Reporting*. When the applications starts it sets up the *context* and opens a socket on port 5684 (section 8.6 [3]) waiting for incoming registration messages. The LWM2M APIs also offers the opportunity to define a callback to be invoked when a response message is received. In our case the callback simply parses the response payload and prints it in the standard output accordingly to its format (TLV, JSON, Plain Text). Another callback, *monitorCallback*, is triggered when the client registers, updates or de-registers its subscription to the Server.

Client

Likewise, the test Client application provided by Wakaama implements several LWM2M interfaces (Data model, *Registration*, *Device Management* and *Information Reporting Interface*) to be used in our prototype. When it starts it first initializes the *context* with the defined objects and information about servers and their security requirements, then it tries to open a connection towards a predefined server and starts the registration process. The test Client application provides three objects: Device object, Firmware object (both defined by the standard [3]) and a Test object, used as an example, with the purpose of storing a single integer value. For this last

object we defined a routine that updates with a random number the value of the resource in all its object instances, in this way we can simulate the state changes that happens in real devices.

Proxy

The proxy, as defined in Section 3.4, is an entity that can act both as Client and as Server. For this reason the Proxy implements both interfaces by defining the *LWM2M_CLIENT_MODE* and *LWM2M_SERVER_MODE* constant for conditional compilation. The Proxy offers a terminal interface with access to both the client and the server API.
The implementation uses a single UDP socket on port 5683 for receiving messages for the Client and Server interfaces, it adds a Server to the *context*, to which it will register later, and it also set objects that is able to support, object that we will explain in next sections. After the initialization the Proxy application tries to connect and register to all preconfigured Server, waiting then for any incoming packet or any input from the terminal console.

4.4 Group Object

In order to implement a new object we need to define the callback functions for the *lwm2m_object_t* as well as the parameters listed in Section 4.2.2. For the Group Object we have to follow the Template defined in Table 3.1 and we can begin by assigning as Object ID, for example the number *26890,* to identify the Group Object. Since we have 6 Resources, the value of *MaxResourceID* is *5* (the first resource has ID 0).

Regarding the Instances we said in Section 3.5.1 that every instance matches a group, so we need only to store the list of members and requests sent. To do so we use the structures showed in Figure 4.3. *Next* and *shortID* attribute are needed to build the linked list of instances, *clientList* is a pointer to the head of the Client list belonging to the group. We need only the ID because the Proxy already has a list of the registered Clients, thus we need to store only a reference to them, that is the ID of the Client.
Lastly we use the structure *prv_fwStructure*_t to store information about the requests sent from the Server to the Group, in particular we store how many responses are yet to be received (attribute *resp2bRecv*), the responses received (attribute *FW_response*) and the destination URI. Next we will see how these structures are used during operations.

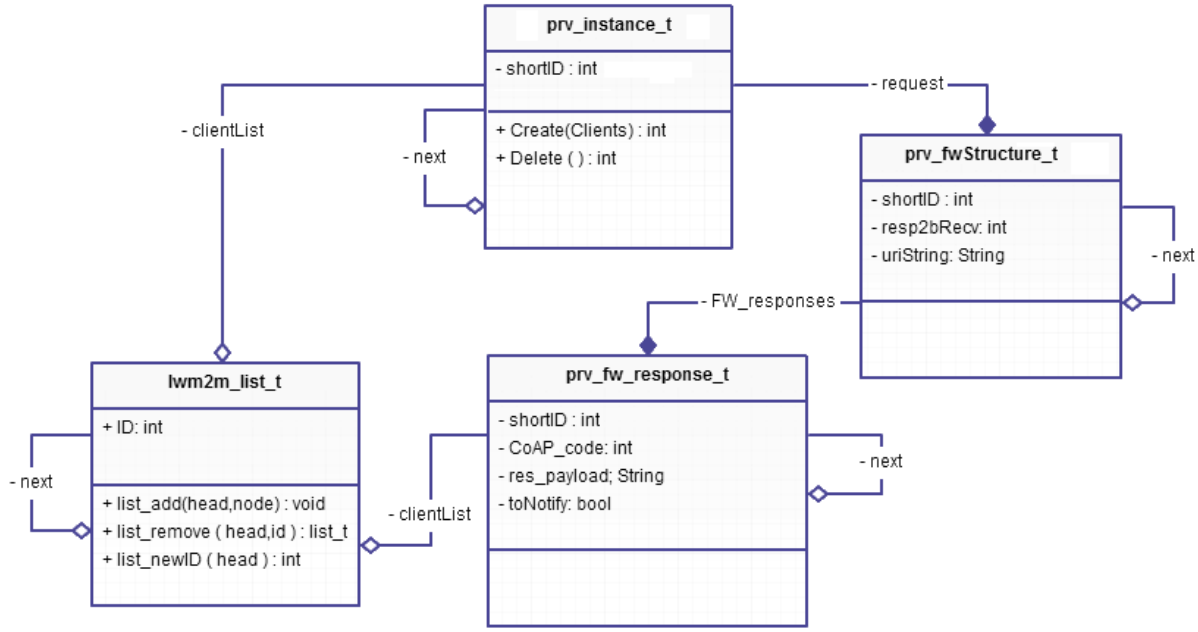


Figure 4.3: Group Object Structures

Group Creation

To create a new Group, as explained in section 3.5.2, the Server must send a Create message to the Group Object in the Proxy with list of URN of Clients that will become members of the Group in the payload of the Create request. From the implementation point of view this translates in the LWM2M Server sending a *CREATE* message (CoAP POST) with the GroupObject ID (that we assumed to be 26890) as destination URI and optionally the ID of the Object Instance that we want to create (if it does not already exist). URNs must be concatenated in a single string and separated by a space.

Once the Proxy receives the *CREATE* request, the *createFunc* callback of the GroupObject is invoked and it will parse the list of Client URNs as well as check that all of them are of registered Clients of the Proxy (Figure 4.1). For every checked Client, a node with the same "Client internal ID" of the proxy is inserted in the list of member of the group (*ClientList* attribute). A control on the Instance ID is also performed if it was provided, that is to check if the InstanceID already exists. If any error occurs, the matching error code (401, 404, 500, etc.) is returned from the function, otherwise it will allocate a new *_prv_instance_* structure and will initialize the available parameter. The following code is an initialization example:

```
newGroup = (prv_instance_t *)malloc(sizeof(prv_instance_t));
newGroup->shortID = lwm2m_list_newId(objectP->instanceList);
newGroup->clientList = LWM2M_LIST_ADD(newGroup->clientList, ClientID);
newGroup->requests = NULL;
```

The result of the operation (success or failure) is sent back to the Server as the matching CoAP response code.

Adding Members

In order to add one or more members to a Group, the Server must send an *EXECUTE* message (CoAP POST) to the Proxy with URI composed of GroupObjID, InstanceID of the relative Group and as Resource ID, the ID of the resource *Add Member* (which is 2 in our case), i.e. "/26890/1/2". As argument of the *Execute*, the server must provide the URN of Clients to be added to the Group as a String, concatenated by a space. After having parsed the CoAP message, the Proxy invokes the *executeFunc* Callback of the GroupObject that will process the Clients URN in the same way of a create request: it will check if the Clients are registered and then add it to the members of the Group by adding a node (for each Client) with the same "internal ID" to the *ClientList*. If any error occurs, the relative CoAP response code is returned from the function.

Forwarding Messages

When the Server sends a command to all Group Members it must perform an *Execute* to the *forward* resource with the serialized command, the URI as well as the arguments to be forwarded as payload (refer to Section 3.5.5).
Once the Proxy receives the message, the function *executeFunc* is executed. This function will parse the payload and it will test the arguments in order to verify that:

1. the command (represented by a char) is one in the set {R,W,E,C,D} (*d* and *w* commands are not implemented in Wakaama and they are not needed for our prototype);
2. the URI provided is syntactically correct (for the command provided);
3. any needed argument is provided;

If all these conditions are satisfied, these data about the request will be stored in a new *prv_fwStructure_t* instance.

```
newRequests = (prv_fwStructure_t *)malloc(sizeof(prv_fwStructure_t));
newRequests->shortID = lwm2m_list_newId(newGroup->requests);
newRequests->uriString = uri2string(uriP);
newRequests->operation = operation;
```

```
newRequests->resp2bRecv = numOfMembers;
newRequests->FW_responses = NULL;
newGroup->requests = LWM2M_LIST_ADD(newGroup->requests, newRequests);
```

After storing this information, the LWM2M API matching the received command is invoked for every member of the group. As argument of the API will also be provided: the internalID of the corresponding Client; a callback function to process the Client response and a reference to request data just allocated.

Invoking the API means that a transaction will be registered in the Proxy, waiting for the Client's response. Lastly, the Proxy sends an aknowledge message to the Server to let him know that the command was forwarded and that the result will be available in resource *FW Result* (ID 5). As we commented in the previous chapter (section 3.5.4), the response to a group request will have the following header:

```
{"u": "/1024/10/1", "op":"R","missing":0, "r":[]}
```

which is, as we can see, a simple serialization of the thata stored in the *prv_fwStructure_t* structure.

When a Client response is received from the Proxy (or the timer times out), the transaction is removed and the callback function invoked. The callback decreases the number of response to be received by one (*request-> resp2bRecv*) and adds a node to the request response list (*FW_responses*) with the information of the response: sender ClientID, CoAP response code for the message and Payload (if any). This information will be returned as a JSON in the "r" array of the response "header".

```
newResponse->clientID = ClientID;
newResponse->CoAP_code = response->code;
newResponse->payload = response->payload;
requests->response = LWM2M_LIST_ADD(requests->response, newResponse);
```

Removing Members

Similary to the addition of members, in order to remove one client from a Group, the Server must send the an *EXECUTE* message (CoAP POST) to the Proxy. This request message will have destination URI composed of GroupObjID, InstanceID (of the Group) and, the ID of the *Remove Member* resource as Resource ID (which is 3). As an argument of the *execute* command, the list of URNs of the Clients to be removed from the Group must also be provided as a String, with every URN separated from the next by a space.

When the Proxy receives this message, it will check if the URNs provided in the *Execute* match to the URN of the Group Members and if this happens, the node corresponding to the those Clients will be removed from the *ClientList* of the specific instance. If the operation fails, the corresponding CoAP response code is returned as error.

Group Removal

In a similar way to the *CREATE* process, when the Server wants to remove a Group it must send a *DELETE* (CoAP DELETE) message to the Proxy, with the URI of the instance to the delete, no other arguments are needed. When the message is received and parsed from the Proxy, the object callback *deleteFunc* is invoked. This function iterates through the list of requests of the group (*request* attribute) and removes all the data relative to its response messages (*FW_responses* attribute) and then the data of the request itself (*request* attribute). Any responses received from clients, belonging to an ongoing Forward operation, will be ignored. Once the request list is empty, the function removes all members of the group (*ClientList* attribute) and then removes other Instance data. Once all operations are completed without errors, a response message with code *"202 Deleted"* is sent to the Server, the corresponding 4.xx or 5.xx error code otherwise.

4.5 RegistrationPolicy Object

Proceeding in the same way done for the Group Object, to implement the object template of the RegistrationPolicy Object (Table 3.3), that will be used for indirect group management, we need to define the callback functions of the *lwm2m_object_t* as well as the other parameters. As Object ID we chose *21995* and *maxResourceID* is 3.
In this Object, every instance matches a Group. A list is used to store instance information and the *clientList* resource is a pointer to the head of the Client list that are members of the group. The object structure has also a list for storing policies for a particular Group/instance. Properties and relation of object's structures are showed in Figure 4.4.

Group Creation

For this object, the information needed to create new instance is the list of criteria that clients must satisfy to belong to particular a group. The *CREATE* command (CoAP POST)

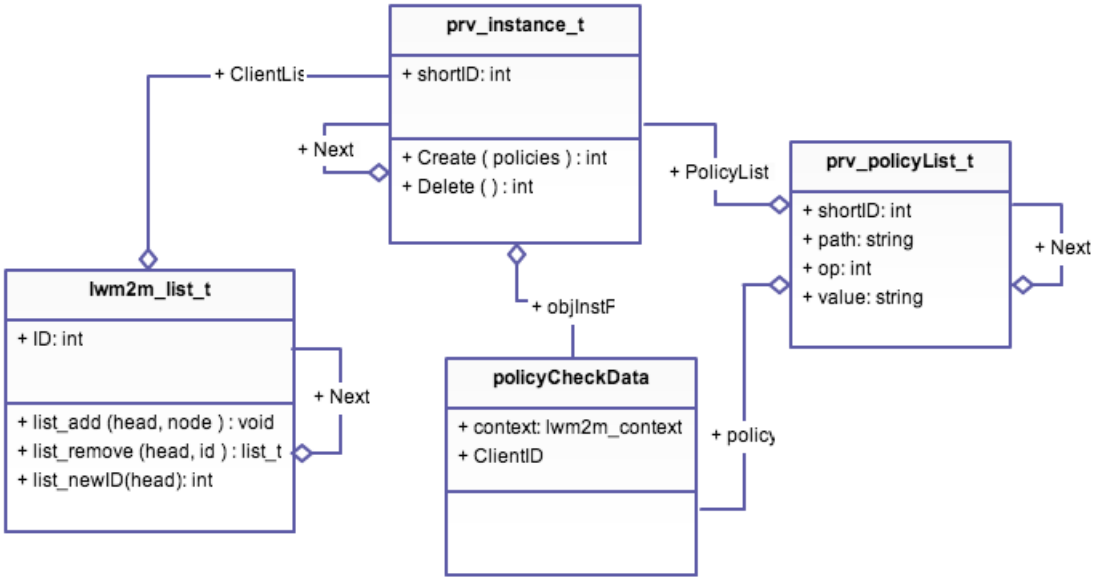


Figure 4.4: RegistrationPolicy Object Structures

sent from the Server, with URI "/21995", to the Proxy must contain a string representing the policies in the JSON format defined in 3.5.4. For reference an example of policy is {"u"="/1024/10/1","o"=>,"v"="5"}.

When the Proxy receives the *CREATE* request, the *createFunc* callback of the RegistrationPolicy Object is invoked. This function will parse the list of policies and check if all of them are valid. If one of them is not, due to incorrect format or operation not supported, a CoAP 400 (BAD REQUEST) response code is returned.
If all the policies are valid, each one is stored in a *prv_policylist_t* node and added to the instance's list of policies. In particular, the information saved are: URI of the resource to test ("u" attribute in the JSON), reference number of the operation (0 for *less than,* 1 for *equal*, 2 for *greater than*) and the string representation of the value to be compared.

If no error occurred, a new *prv_instance_t* structure is allocated and initialized, the matching error code (401,404, 500, etc.) is returned otherwise. Following, an exaple code to initialize a group with the policy is showed :

```
newGroup->shortID = lwm2m_list_newId(objectP->instanceList);
newGroup->clientList = LWM2M_LIST_ADD(newGroup->clientList, ClientID);
newGroup->policies->next=NULL;
newGroup->policies->path =strdup("/1024/10/1");
newGroup->policies->op =2; //greater then
newGroup->policies->value=strdup("5");

objectP->instanceList = LWM2M_LIST_ADD(objectP->instanceList, targetP);
```

The result of the operation (success or failure) is sent back to the Server as the matching CoAP response code.

Policy Routine

When a Client registers, the *Policy Check* process is started. This process is performed for every object instance defined and, as explained in Section 3.5.4, it consists in evaluating if every criteria of a policy is satisfied by the Client's Resources. From the implementation point of view this translates in two functions:

```
start_group_policy_check( contextP, clientP)
policy_check_response(clientID, uriP, status,
 data, dataLength, userData)
```

The first is used to setup the needed structures to start the *policy check* process, the later is used as callback for the first and to iterate the process for the next criteria of the list (if any).

In particular, start_group_policy_check() is called from *monitorCallback* (one of Wakaama's function, see Section 4.2) when a Client successfully registers and a reference to the client structure is passed as argument. The function then fetch all the instances of the *registrationPolicy* object and for every one of them it performs the following operation:

1. It check that the client is not already a member of the group (it may happen if the client sends a duplicate registration message);
2. If it is not already a member, the function fetches the first criteria of the *policyList* and retrieves the Resource URI to be evaluated;
3. It sets up a new *prv_policyCheckData_t* element where will store a the clientID, a reference to the object instance, a reference to the PolicyList and a reference to the *lwm2m_context*.
4. It sends a READ command to the client under evaluation with destination URI equal to the one fetched from the first criteria of the *policyList*. It also sets the *policy_check_response()* as callback function and the previously created *prv_policyCheckData_t* element as parameter for the callback.

When the response of the *READ* command is received, or the timer times out, the function *policy_check_response()* is executed and the second part of the process begin:

1. The function first checks if there is any data (payload) available in the response (the callback is executed also if the client returned error or the request timed out).

2. if data is available, the *prv_policyCheckData_t* element passed as argument is used to retrieve information about operation and value to be used for the comparison with the resource's value.

3. If the operation is "<" or ">" (*less then* or *greater then*) the received value and the one stored in the *policyCheckData* element are converted from String to an integer in order to correctly perform the comparison. A normal String compare is performed for the "=" (*equal*) operation.

4. If the evaluation is not successful, which means that the Client's Resource does not satisfy the criteria, the function frees any allocated data and returns, stopping the *policy check* process.
   If instead the criteria is satisfied is possible to proceed in two different way:

   (a) If there are any other criteria to be evaluated, the *policy* attribute of the *policyCheck-Data* element is changed to point to the next policy element of the list. The URI to be evaluated for following criteria is used as URI for a new *READ* command (COAP GET) that will be sent to the client, again with *policy_check_response()* as callback and the *policyCheckData* element as argument.

   (b) If there are not, all the criteria for that Group/Instance are satisfied, thus the Client can be added to the member of the Group by creating and adding *clientList* a new *lwm2m_list_t* node with *ID* attribute equal to the the client internal ID.

Update Trigger

Another functionality provided by the Registration Policy object is the opportunity to verify if all member still satisfy the Policies using the Update Trigger. To activate this process the Server must send an *Execute* command to the *Update* Resource (ID 3), no arguments are required. When the Proxy receives this request it will fetch the list of registered Clients and reset the membership, that is removing every client from the *ClientList*. After this, for every Client in the Registered List the *start_group_policy_check()* function is executed, starting the policy check process described in the previous subsection and that will verify if the criteria of the policies are still satisfied, then the client is added again to the group, or not.

Group Removal

When the Server wants to remove a Group it sends a *DELETE* (CoAP DELETE) message to the Proxy, with URI of the instance to the delete. No other arguments are needed. When the message is received and parsed from the Proxy, the object's callback *deleteFunc* is invoked. This function iterates through the list of Group's policies (*policies* attribute) and removes all its data. Once the list is empty, the function removes all the member of the group (*ClientList* attribute) and then remove the other data of the instance. Once all those operations completes without errors, a response message with code *"202 Deleted"* is sent to the Server, the corresponding 4.xx or 5.xx error code otherwise.

4.6 GroupObserve Object

Similarly to the *Group* Object, to implement the *GroupObserve* Object we have to follow the Template defined in Table 3.5. As Object ID, for this object we choose randomly the number *41188*, and since we have six Resources, the value of *MaxResourceID* is *5*.

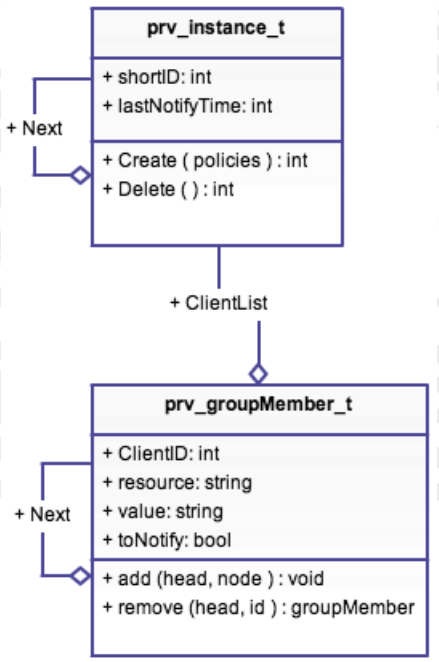


Figure 4.5: GroupObseve Object Structures

Regarding the Instances, also in this case every instance matches a group, so all group data are stored as instance, for which we use the structures showed in Figure 4.5. *Next* and *shortID* attribute are needed to build the linked list of instances, *clientList* is a pointer to the list of

information of every client, in particular the internal ID of the client (*clientID*), the URL of the observed Resource (*resource*), the last know value (*value*) of the observe resource and a flag (*toNotify*) in order to track if the latest *notify* received from the client was already sent to the server in an *Aggregated Notify* or not. Information about the last time an aggregated notification was sent to the Server is also stored in the instance (*lastNotifyTime*). Next we will see how those structures are used during operation.

Group Creation

To create a new Group, the Server must send a Create to the *GroupObserve* Object in the Proxy and a the list of pairs {URN, Resource} as payload, which are the URN of the Clients and URI of the resource that must be observed. This translates in the LWM2M Server that sends a *CREATE* message (CoAP POST) with destination URI the GroupObject ID (41188) and eventually the ID of the Object Instance that we are trying to create (if not already exists). In the Payload every pair URN-URI must be concatenated by a space in a single string.

Once the Proxy receives the *CREATE* request, the *createFunc* callback is invoked and it will parse the list of pairs (Client URN, Resource) and for every pair it will check if a client with the provided URN is registered List and then create a a new list of *prv_groupMember_t* nodes in which every attribute have the following meaning and value: *ClientID* is the internalID (in the Proxy) of the registered Client; *Resource* is a copy of the Resource provided in the request that should be observed; *Value* is the latest know value of the resource to observe, so it is set to NULL in this phase; *toNotify* is a flag used to tell if the latest *Value* was sent to Server and it is set to 0. If one of the clients is not registered, the procedure stop (removing all the nodes of the new list) and return a "400 - bad request" error. An example of initialization is the following code:

```
newGroup = (prv_instance_t *)malloc(sizeof(prv_instance_t));
newGroup->shortID = lwm2m_list_newId(objectP->instanceList);
newMember = (prv_groupMember_t *)malloc(sizeof(prv_groupMember_t));
newMember->resource = clientResource;
newMember->clientID =prv_find_clientIDByName(clientURN);
newMember->value=NULL;
newMember->toNotify = 0;
newGroup->clientList = LWM2M_LIST_ADD(newGroup->clientList, newMember);
```

If all the clients are registered and no error occurs then, for every client/node in the list, an Observe request (COAP GET + Observe option) to the respective Client's Resource is made. The *ObserveCallback(),* function with a reference to the object Instance is also set as callback

of the Observe request in order to process subsequent notifies. When the Proxy receives notification from a client regarding an observed resource, the callback takes the payload of the notify and store it in the *Value* resource of the corresponding *clientList*'s node and it sets to 1 the *toNotify* flag. In this way information is made available to the Server through the Proxy's *value* resource. If the observe request did not return any error, a new *prv_instance_*t structure is allocated, initialized and added to the instance's list of the object, a CoAP 204 Response code is also returned to the Server. If instead any error occurred: the matching CoAP response code (401, 404, 500, etc.) is returned.

The creation of a GroupObserve does not mean that the server will be notified of new values from clients, in fact as explained in Section 3.5.5, the Server needs to perform an Observe request on the *values* resource of the GroupObserve Instance just created. In this way the Server will be able to receive an aggregated Notify when new value of observed resources of clients are available. However, to avoid to receive multiple aggregated Notify (every time a new update is received from clients), two condition are introduced to limit the frequency of Notify to the server. Details are explained next.

Group Observe Routine

As we said previously, even if the Proxy receives a notify from an Observed Client that belong to a GroupObserve, the new information is only made available to the server through the *Value* Resource. For the Server to receive an aggregated notify, one of the following conditions must be met:

1. All Clients members of the GroupObserve have sent a new notify since the last aggregated notify to the Server;
2. At least one Notify was received since last aggregated notify to the Server and no other notification was received after it for a certain amount of time;

To verify if one of these condition is satisfied, the function *group_observe_routine()* is periodically executed in the Proxy. For every GroupObserve Instance, this function iterates through all *ClientList'*s node in order to set two variable: *allUpdated*, which is set to 1 if all the node have *toNotify* equal to 1 , and *anyUpdate*, witch is set to 1 if at least one node has the attribute *toNotify* set to 1. It is possible to think of *allUpdated* as the result of the logic AND between all the *toNotify* attribute of *ClientList*, instead the *anyUpdate* is the logic OR.

If *allUpdate* is set to 1 (first condition), or if *anyUpdated* is 1 and the instance attribute *lastNotifyTime* is less then current time minus a certain amount (second condition) the Wakaama API *lwm2m_resource_value_changed()* is called for the *values* Resource. This API will send a Notify with the latest value of the Observed Resources (stored in *values*) to every Server observing that resource. Then all the *toNotify* attribute are set to 0 and the *lastNotifyTime* update with the current time.

Adding Members

In order to add one or more members to a Group, the Server sends an *EXECUTE* message (CoAP POST) to the Proxy with URI composed of GroupObserve ObjectID, InstanceID of the relative GroupObserve and as Resource ID, the ID of resource *Add Member* (which is 5 in our case), i.e. "/41188/1/2". As argument of the *Execute*, the server must provide the URN of Clients to be added to the Group and the Resource to be Observe concatenated by a space.

When the Proxy receives the CoAP message, it will parse the message and it will invoke the *executeFunc* Callback of the GroupObserve Object which will process the Clients URN and Resource URI in the same way of a *Create* request: it will check if the Client is registered and then add it to the member of the Group by adding a node with internal ID and Resource URI while other attributes are set to 0 or NULL. After this an Observe Request is performed on the Client's resource provided in the request. If any error occurs, the relative CoAP response code is returned from the function, otherwise a "205" response code is returned.

Removing Members

To remove a member from a Group, the Server must send an *EXECUTE* message (CoAP POST) to the Proxy with URI composed of GroupObjID, InstanceID of the relative Group and as Resource ID, the ID of resource *Remove Member* (which is 6) and as argument, the URN of the Client to be added to the Group in a String format. When the Proxy receives this message, it will look for the corresponding *internalID* and then it will look for a node with the same *shortID* in the *ClientList* of the corresponding object instance. If it is found, it is removed from the list (de-allocating any related resource), the corresponding CoAP response error is returned otherwise.

Group Removal

Likewise to the *CREATE*, when the Server wants to remove a Group it must send a *DELETE* (CoAP DELETE) message to the Proxy, with URI of the instance to the delete. No other arguments are needed. When the message is received and parsed from the Proxy, the object callback *deleteFunc* is invoked. This function iterates the *clientList* and for every node it perform a CancelObservation on the Resource of the ClientID, which is used to delete the resource and client from the Wakaama internal *ObservedList*. Once all these operations are completed without errors, a response message with code *"202 Deleted"* is sent to the Server, otherwise the corresponding 4.xx or 5.xx error code .

4.7 ObservedMonitor

As we explained in section 3.5.6, the ObservedMonitor Object is used to monitor all the observation currently active on a Client, (but also client-side of a Proxy). Since every Object Instance matches an Observe on a Resource, no further structures where added. Instead a new attribute, the *maxObserver* attribute, was added to *observedList*, the Wakaama internal list where information on current observation is stored. Since this Object refers to an internal status of the Client that is automatically modified from other factor, the implementation of a callback for the CREATE and DELETE command is not needed. The only functions to be implemented, following the template 3.6, are the *readFunc* callback and the *writeFunc* callback.

When the Server sends a READ request (CoAP GET) for the *Observed URI* Resource (ID 0), after parsing the message the Proxy executes the *readFunc* callback. This looks for a node of the *ObservedList* with attribute *shortID* corresponding to the value of the InstanceID in the request URI. If such node exists, the observed Resource URI is returned to the sender of the request as a string. Otherwise the corresponding CoAP response code is returned.
Regarding the *ObserversID* the same process just described is performed, and once the node corresponding to the InstanceID is fetched, is possible through it to obtain the list of *watchers* (that is the list of Servers observing that client Resource). The information for each "watcher" that will be returned to the requesting Server are *host* and *port*, in the format "[host]:port" in order to comply also with any IPv6 address.

Lastly the *maxObserver* Resource is a single number that is set to 100 by default and that indicates the max number of observers for a specific resource. If any Server tries to perform the 101st observe request, this one will fail even if all the required parameter are correct, returning then a CoAP 405 error (method not allowed).

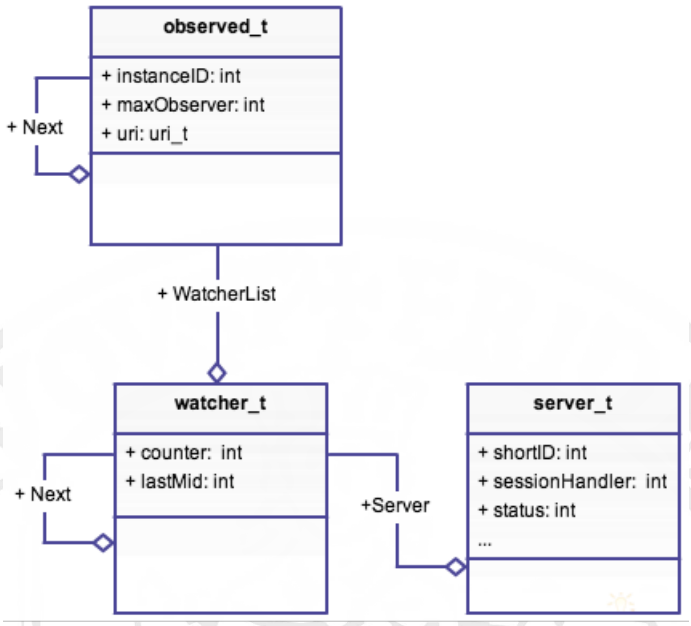


Figure 4.6: ObserveMonitor Structures

In the case of the write operation, the Server must send a WRITE request (CoAP POST) with the new value of the *maxObserver* as payload of the request. The Client, after parsing the message, will trigger the *writeFunc* callback which will first test if the payload of the WRITE is a number and then will fetch the node of the *observedList* corresponding to the value of the InstanceID in the request URI. If it is found, the value is updated and a successful CoAP response code is returned to the Server; the CoAP response code corresponding to the error is returned otherwise.

4.8 ProxiedDevices Object

Like the previous object, the ProxiedDevices Object takes advantage of its existing internal structure to provide its functionalities. In particular, since this object refers to Clients registered to the Proxy, the structure used refers to the Wakaama internal *ClientList* showed in Figure 4.1 and detailed in Figure 4.7, which also store a copy of objects supported from the client. Following the template showed in Table 3.7 we have to implement the *readFunc*, *writeFunc*, *execFunc*; the Create and Delete callback are not needed since the creation and removal of object instances depends only from the Proxy and cannot be "forced".

For any request from a Server the first control performed is verifying if the request URI has an InstanceID corresponding to an *internalID* of a registered client. If such correspondence exists, then the Proxy tries to serve the request, invoking the specific command callback, error

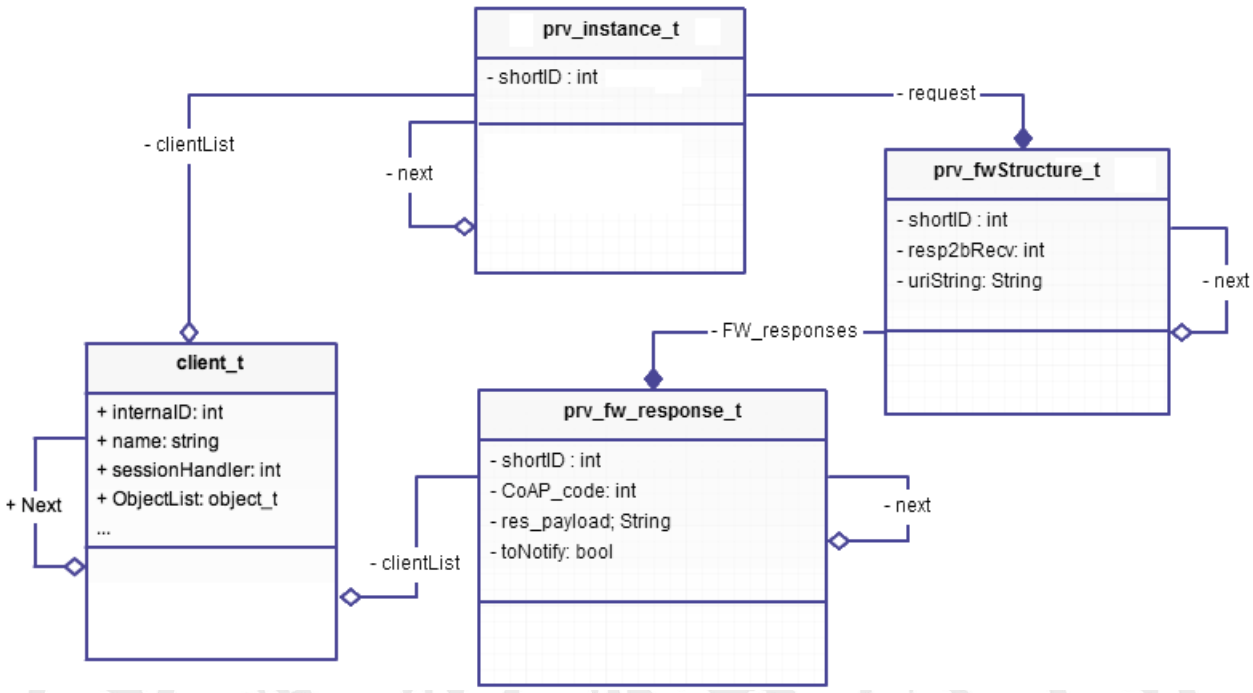


Figure 4.7: ProxiedDevices Structures

is returned otherwise.
If the server sends a READ request for the first resource of the ProxiedDevice template, the *readFunc* callback will fetch the *ClientList* node and return the InternalID of the node. For the second Resource (*ClientID*, ID 1) the *readFunc* will return the URN of the client, stored in the *name* attribute of the *ClientList* node. For the third Resource (*objectList*) the information is retrieved returning a serialized version of the List *objectList*, attribute of the *clientList* node. The serialized version will comply with with the CoRE Link Format [26] in order to facilitate reuse of any parsing function that is used in the registration phase of LWM2M (see section 2.4.3.2).

Regarding the latest two resources of the object template, they are used in the same way of Resources *Forward* and *Forward Response* of the *GroupObject* explained in Section 4.4. The only difference here is that there is only one client, so the format of the *Forward Response* is simpler. An example for a Read request is:

```
{"q":"R  /1024/10/1", "r":{"c":204, "v":1024 }}
```

As you can see only one field is used for query information (command type and URI) and another one for the response, where the CoAP response code and the value are returned.

Chapter 5

Discussion/testing

In Chapter 3 we listed the requirements to obtain a Group Communication in Lightweight M2M. Here we will show which of those requirements are met from our prototype and how our system behave in common use cases. Lastly we will show results of performance measurements conducted while testing the prototype and how the introduction of a Proxy affects the performance of a client-server system.

5.1 Functionality Evaluation

To test the functionality of our prototype, common use cases were executed with the prototype. The system used as testbed (Figure 5.2) consisted of two machines: one running the LWM2M Proxy and the LWM2M Server/Manager (Machine A) and the other one running a certain number of LWM2M clients (Machine B). These machines where connected to the same LAN through Ethernet.

5.1.1 Use Case Execution

Let us suppose we have to manage a street lighting system. There are multiple streets and every street has multiple street lights. We assume that every street light has also a light sensor, a connectivity module and the software to act as a LWM2M Client able to receive commands via network (protocols supported by the connectivity module) from LWM2M Server/the Manager, through one or more LWM2M Proxy.

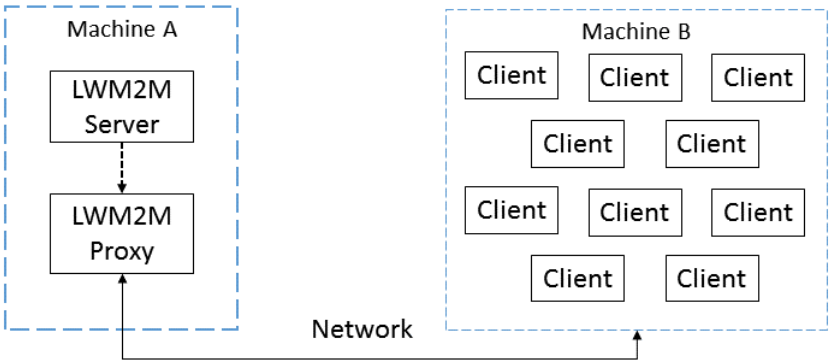


Figure 5.1: System used for testing

Discovery of Group and Members

When turned on, the street lights boostraps and try to register to their local LWM2M Proxy. The Proxy should be already registered to the LWM2M Server/Manager. If the Server does not have already information on the LWM2M Clients, read this information in the object *ProxiedDevices*. Then it will create a single group for all registered street lights, in order to optimize future operations. The corresponding sequence diagram is showed in Figure 5.2.

Group Operation

After having created a single group containing all street lights, the manager may want to group lights in smaller groups, for example by street (one group per one street). To do so, it must collect information on the location of every light (which could be embedded in the device or be provided by a GPS module or be already known to the Server), thus it could perform a request through the GroupObject, receive and elaborate the information and then create a new Group for every street.

Since new devices may be added, it was decided to take advantage of the fact that street lights, thus LWM2M clients, share the same properties (therefore same LWM2M Object template) and define some rules to be used by the *RegistrationPolicy* object in order to automatize grouping of new clients. The resulting activity diagram is shown in Figure 5.3.

Information reporting

Another useful feauture of the system is the gathering of data from light sensor and automatically turn on and set the intensity of the street lights. This is performed by creating a GroupOb-

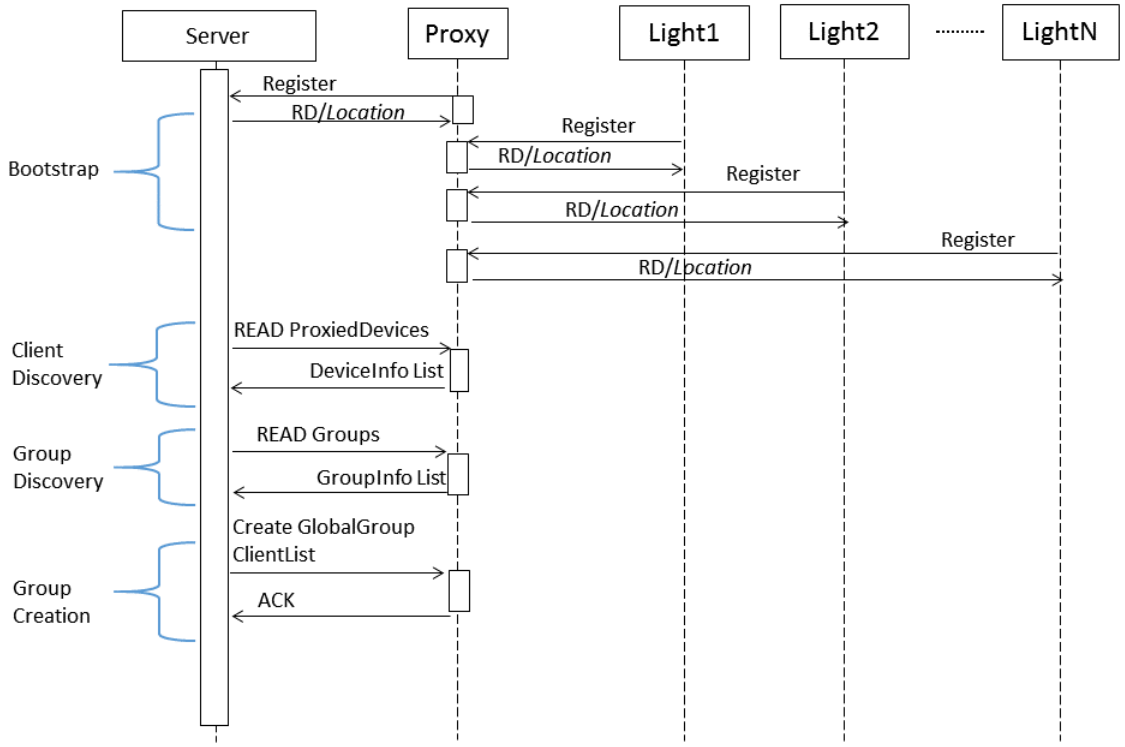


Figure 5.2: System setup and Member Discovery

serve in order to gather data and then, after processing it, sending the group command to set the resulting level of intensity in the street light. The activity diagram showing this interaction is shown in Figure 5.4.

5.1.2 Fulfilled Requirements

The previous use case execution shows that most of the requirements listed in section 3.2 were satisfied. A simple overview is shown in Table 5.1.

For requirement #1, thanks to the decision to match a Group to an Object instance, the prototype is able to provide the *Group Discovery* functionality simply by getting the list of object instance (using UPDATE or READ commands).

Requirement #2, To *Retrieve or Query Group Properties*R, multiple methods are available based on the desired information. For example, to retrieve how often the Proxy should forward an aggregated notification (Section 4.6) it can just read the properties of the *Values* resource of

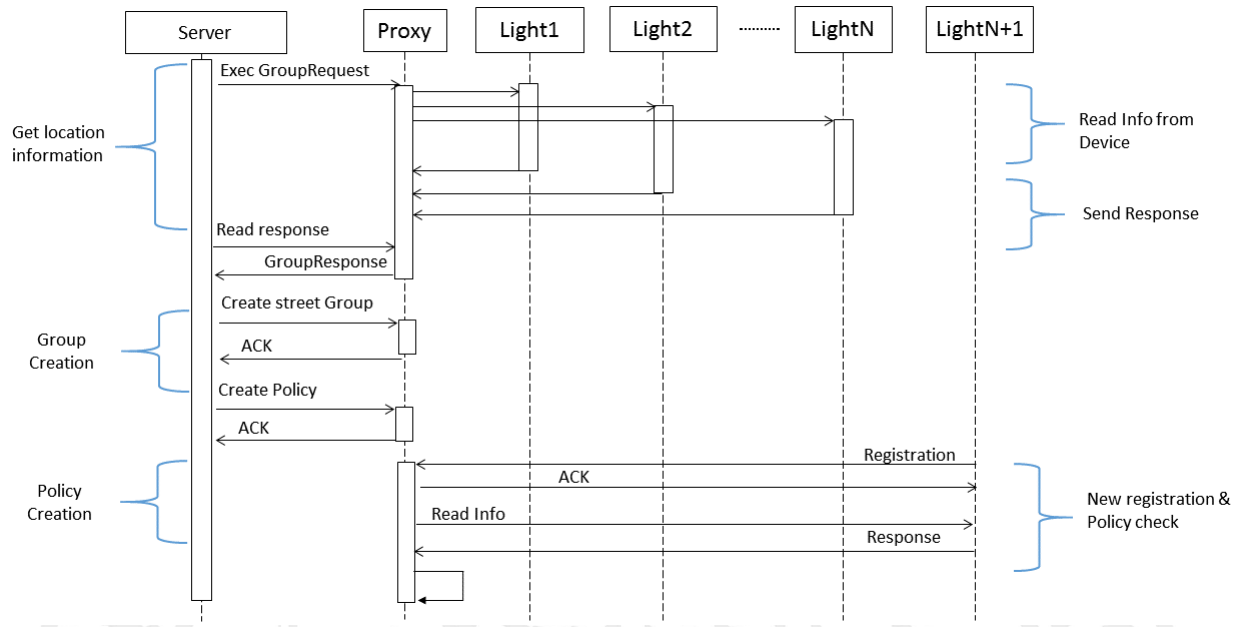


Figure 5.3: Group Operation and Policy Automation

the GroupObserveObject. To know instead how many Clients are connected to the Proxy, it is possible to Read how many instance of the ProxiedDevices Object there are.

Requirement #3, *Creation and Removal of Groups*, can be performed by simply creating Group Object or GroupObserve Object Instances (Section 3.5.2).

Requirement #4, *Addition and Removal of Group Member* respectively to and from a group, is satisfied in two way:
1) members can be added manually by performing an EXECUTE command on the specific resource of the group object (Tables 3.5 and 3.4);
2) dynamically by defining a policy in the RegistrationPolicy Object.

The enumeration of Group members, also in *Requirement #4*, is performed by simply reading the corresponding *ClientList* resource available in both the Group Object and the Group Observe Object. Furthermore, the ProxiedDevices Object allows to verify which client has an active registration to the Proxy.

Requirement #5, *Operation Mapping* requirement is met through the Group Object and Group Observe Object. The first provide an *Device Management* Interface (Section 2.4.3.3) for Groups through the *Forward* and *Forward Response* Resources (explained in 4.4), the latter instead provides an *Information Reporting* Interface for Groups through the Creation of GroupObserve Instances and the *GroupObserve Routine*.

For requirement #6, *Optional Reliability*, we found that it is not really a compulsory requirement. Since we decided to comply with the LWM2M protocol, by design its request messages

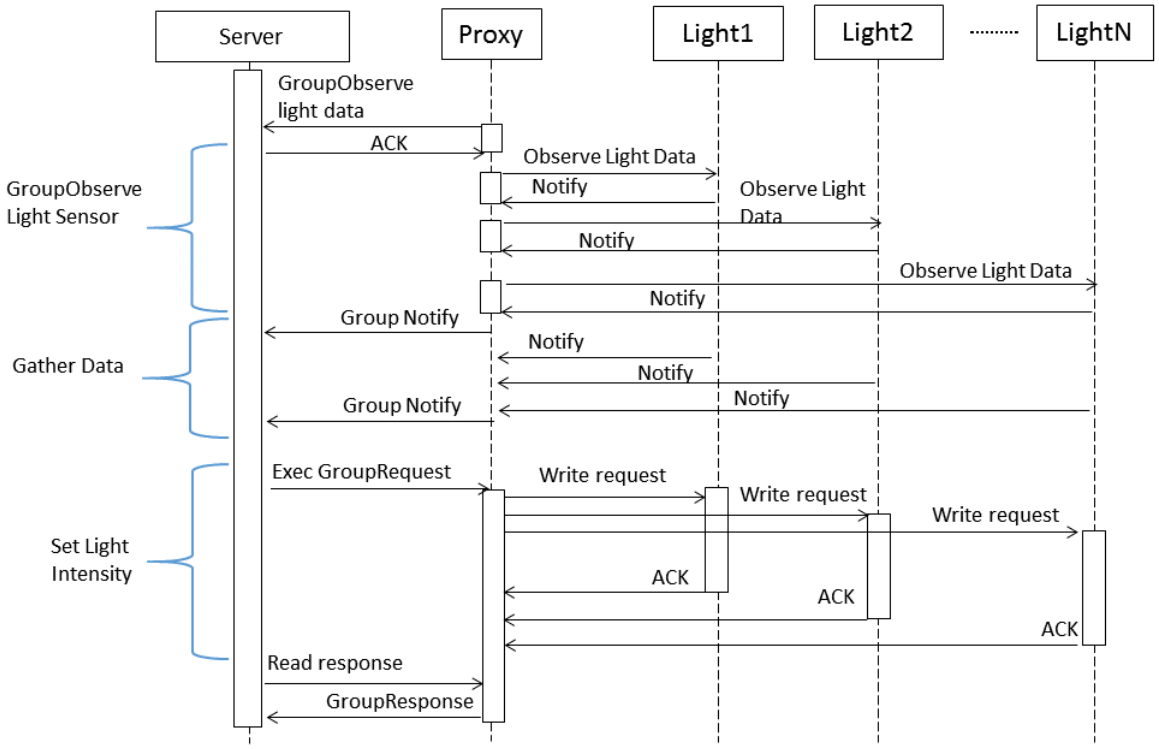


Figure 5.4: Group Information Reporting

uses always CoAP CON as message type (see Section 2.3), this guarantees that the Server is always informed about the delivery status of messages, thus the communication is always reliable, even for Groups.

Requirement #7, *Access Control Primitives* is partly satisfied. In Section 3.2 we stated that system requires that "*only authorized entities can manage groups and perform operation in/on them*". Since a Group is also an Object Instance, the Access Control system of LWM2M [3, section 7.3] guarantees that only the creator of an Object Instance can perform actions on it, unless the creator adds or modifies the access right for another Server. Even so, the problem is that once the Server/Manager is authorized to perform operation on a Proxy, it is also capable of sending messages to every client connected to the Proxy and this is not always advisable. Possible solutions are suggested as future work in the next chapter.

Requirement #8, *Robust Group Management*, is only partly satisfied. We stated that the Proxy should also manage situation like Sleeping node and failing operation but currently, apart from classic input control to validate input from command line and from incoming packets, the prototype only report the most appropriate CoAP response code for problem like server error or unresponsive/sleeping client.

#	Requirement	Satisfied?
1	Group Discovery	Yes
2	Retrieve Group properties	Yes
3	Create/Remove Group	Yes
4	Add/Remove and Enumerate Member	Yes
5	Mapping operation to Groups	Yes
6	Optional Reliability	Not Needed
7	Access Control Primitives	Partly
8	Robust group Management	Partly

Table 5.1: Requirements Table

5.2 Performance Evaluation

We did some measurements in order to investigate the performance of our prototype. In particular, we measured how an increasing number of client would affect memory footprint and throughput/message size of the LWM2M server and then how much improvement would provide the introduction of Proxy in the system. Result of these measurements are presented in this section.

5.2.1 Memory Improvement

Showed in Figure 5.5 there is how memory consumption is affected by the number of clients registered to the Server. As Shown in the Graph the virtual memory required for running the software (even with thousands of clients) is minimum and it increases by just a few hundred of bytes for every two or three hundreds of clients registered.

The same trend observed in the Server were observed also for the memory consumption of the Proxy. Thus, from the memory perspective, a single Proxy is able to handle thousands of clients allowing the Server/Manager to handle only the connection to the Proxy. This is very convenient since it makes possible to run the Proxy even on devices with less capabilities than a normal Server or Cloud system (for example Router/Gateway). Furthermore, since the problem of maintaining the connection is moved from the Server/Manager to other devices, the Manager has more resource to connects and manage a bigger number of devices.

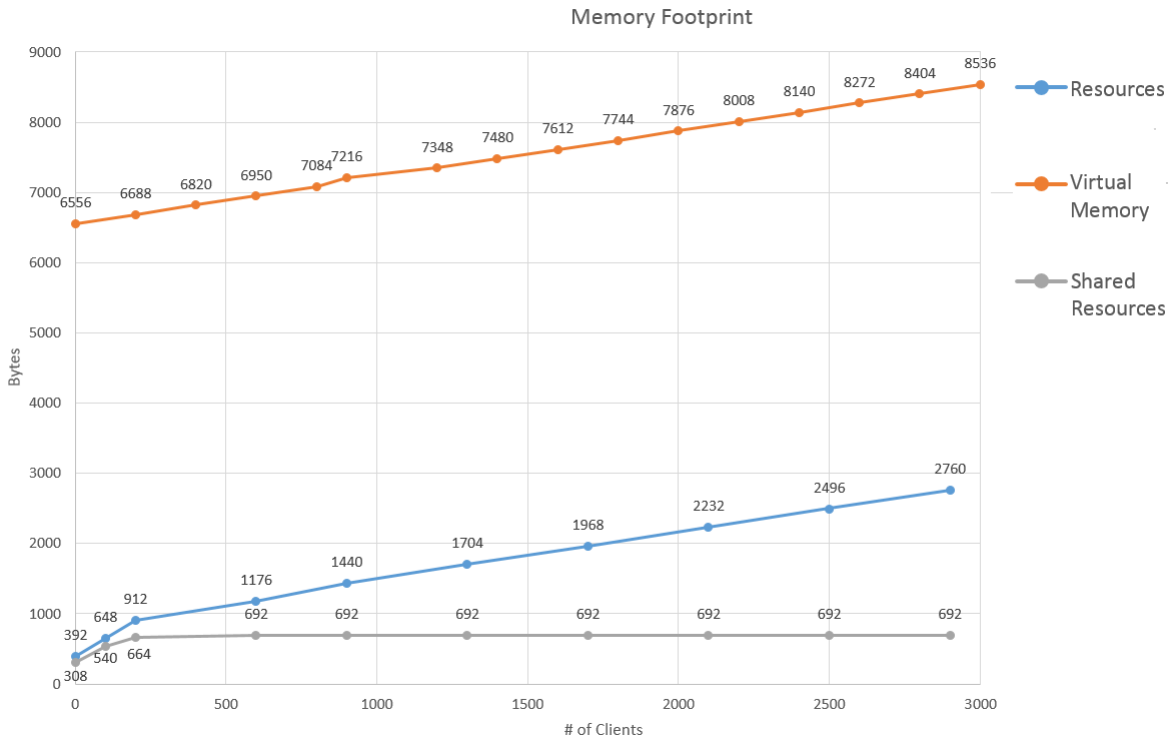


Figure 5.5: Memory Footprints

5.2.2 Protocol Overhead

Multiple tests were run in order to evaluate which was the overhead to perform a complete Group operation, and also to compare it to the overhead of the simple unicast solution, without the Proxy. The tests were divided of 3 phases that addressed the three main operation: 1) Execution of a Read operation 2)Execution of a Write operation 3) Execution of an Observe operation.

We considered as overhead any extra information needed to obtain the Resource value from the Proxy to the Server (since the interaction between Client and Proxy is unicast in any case), thus we have included the headers of lower layers of LWM2M (CoAP, UDP, Ethernet...) and the Header needed to aggregate the resource values in a single response (which we will call Proxy Header).
We have not included the overhead caused by messages needed to create a group, at least in Device Management tests, since these groups could be automatically created through the *RegistrationPolicy* Object (see Section 4.5) or by factory configuration. Also because the cost for Group creation can be spread over multiple consecutive requests and in the long run its overhead became negligible.

Every test was performed running a number of clients spanning from 1 to few thousands, which

was what the machine B was able to run without affecting performance and becoming instable. The payload lengths used where 5 and 250 (1024 for Write). Both number where chosen because they represented the average size of a small and a big payload. In fact we assumed that 5 bytes are more than enough to store even a very big number or a small string, while 250 is suitable to store longer string. In any case, having these two point of reference allow to get the idea of the performance also for value between them.
Following, results of the test and the comparison with the unicast solution are presented.

Read

The first sub-test performed was the Group Read operation. In a normal Server - Clients system, we have that a simple Read request requires 57 bytes. Another 57 bytes are required for the response plus P_{len} bytes for the payload. Thus to perform a Read request and obtain a response are required around 114 bytes plus the size of the payload.

For the Server-Proxy-Clients System instead, as explained in Section 3.5.4, the server first have to create the Group object then it sends an execute request to the Group object in the Proxy that will forward this request to every Client. Once the Proxy receives all the responses and then it will create a single aggregate response that must be read from the Server. Thus, in a complete communication between the Server and the Proxy, the approximate number of bytes exchanged is equals to

$$FwReq + Args + FwAck + ReadReq + ReadACK + ProxyHeader + (RespHeader + CPayload) * NClient \quad (5.1)$$

Where $FwReq$ is the size of the *Execute* Request and requires 57 bytes plus 17 for its *Args*; 60 bytes are required for $FwAck$, the ACK to the *Execute* Request and other 57 bytes are required to send the Read request once the aggregate response is ready.

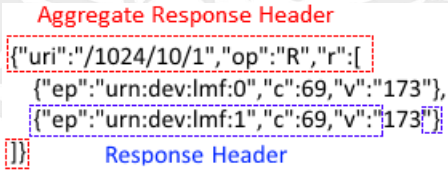


Figure 5.6: Proxy Response structure

This aggregate response is composed of two elements (shown in Figure 5.6): the *ProxyHeader*, which can be at most 51 chars/bytes long, and the Client Response.

The latest is composed of a list of *RespHeader* and *CPayload* for every client response. The first instead is usually around 40 bytes, the second changes based on the length of the Client's Resource requested. In the end we have the following result:

$$Size = NClient * (CPayload + 40) + 242 = F(Nclient, CPayload)$$

We evaluated the previous formula for two values of *CPayload*, 5 and 250, and for an increasing number of Clients. The results are showed in Figures 5.7 and 5.8

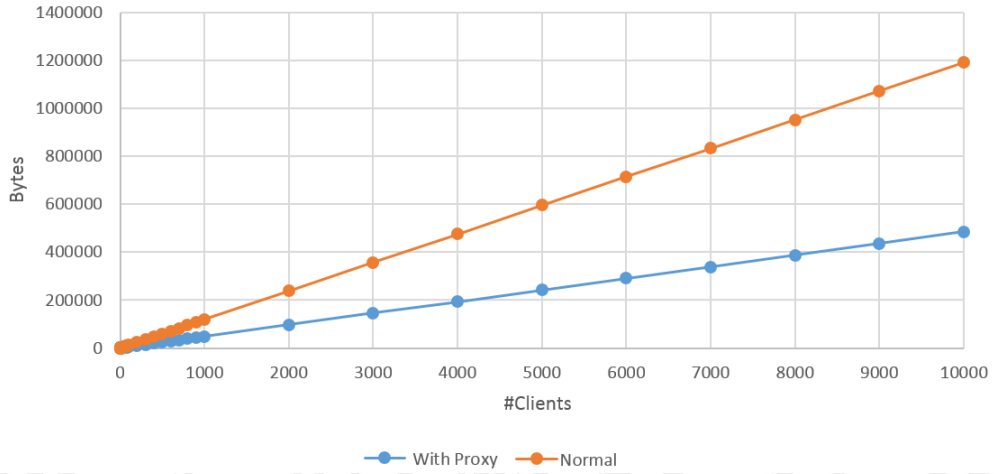


Figure 5.7: Bytes need to perform a Read operation with Payload of 5 Bytes

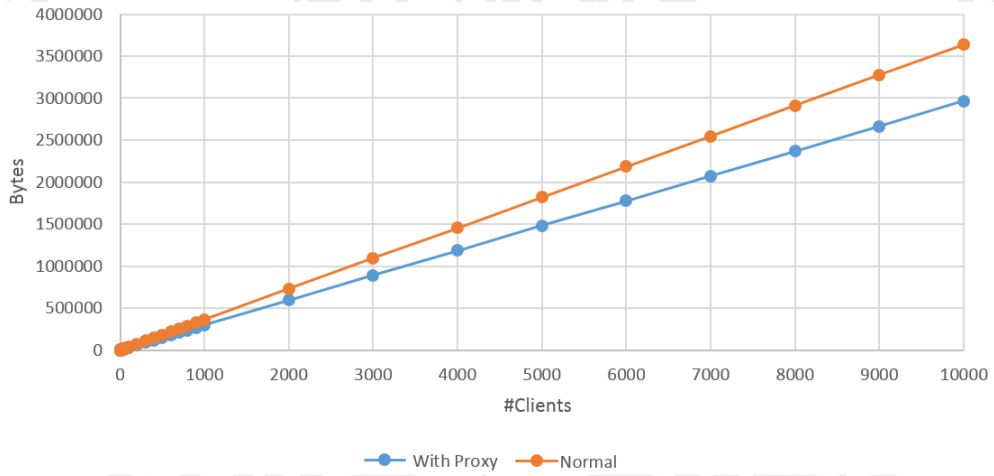


Figure 5.8: Bytes need to perform a Read operation with Payload of 250 Bytes

The results shows that in general, a Group Operation performed via Group Communication Service require less bytes to perform a complete Group Operation in comparison to an unicast solution. Figure 5.7 shows that both solutions have a linear trend for an increasing number of Clients (as expected from the previous formula). From Figure 5.9 is possible to see that only for group with less than five members our prototype has performance less advantageous than the unicast solution, but for a bigger number of clients, the increment goes up to 150%.

The same trend can be observed also if the average Client payload is 250 bytes long (Figure 5.8). In this case however, the improvement that we have with our solution is less then the previous

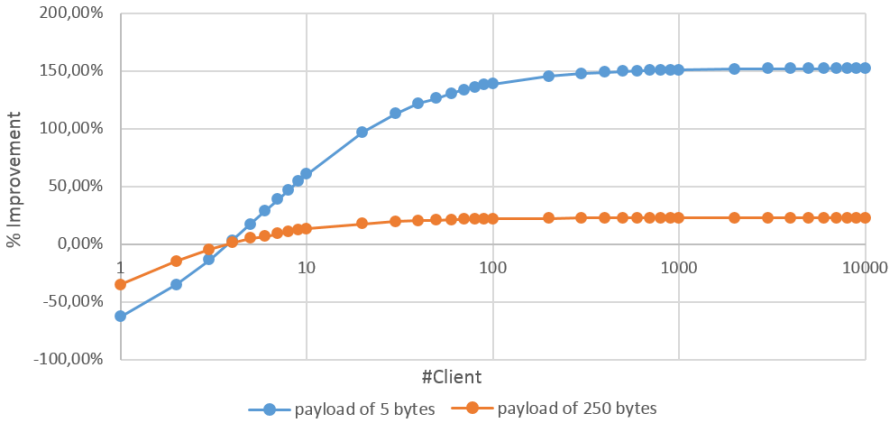


Figure 5.9: Improvement Provided by the proposed solution for Read Operation

case, around 20% as shown in Figure 5.9. Nonetheless it is safe to expect that most of the Read operation on a device will return a response with payload with a lesser length than 10 bytes, being those reading sensor values or device settings.

Small optimization can be applied to further improve the benefit of a Proxy solution in case of the Read Group request. In particular, the optimization explained in Section 4.4 were to include the "value" only if the request is successful; include the "code" attribute only if the request is not successful; use a shorter string as identifier of every client instead of the URI. These optimization affect the $ProxyHeader$ and the $RespHeader$ parameter and they allows to get an improvement almost of 300%.

Write

For the Write operation, the same reasoning applied before can be used also here. In the Server - Clients system, we have that a normal Write request need 57 bytes plus P_{len} bytes for the payload to be written in the client and 57 bytes for the ACK. Thus to perform a Read request and obtain a response are required around 114 bytes plus the size of the payload.

For the Server-Proxy-Clients System, like in the Read case, the Server must send an *Execute* request to the Group object in the Proxy that will forward this request to every Client. Than the Proxy will forward the write request to all the Clients of the Group. In this case the number of bytes exchanged in a complete communication between the Server and the Proxy depends from the formula:

$$FwReq + Args + FwAck + ReadReq + ReadACK + ProxyHeader + (RespHeader + CPayload) * NClient \quad (5.2)$$

In this case, the size of the clients response is fixed while what is changed is the length of the Execute Request, which include the payload that must be written in the clients, and again the number of clients. We evaluated the previous formula for the values of *CPayload* equal to 5 and 1024. Here too we performed the test with an increasing number of Clients (same pattern as before). The two values, 5 and 1024, were chosen because the first represent the average length of parameter to be written in an IoT/M2M device, while the second is a safe guess for very long input (log parameters/configuration or whole object instances) that must be written in the client.

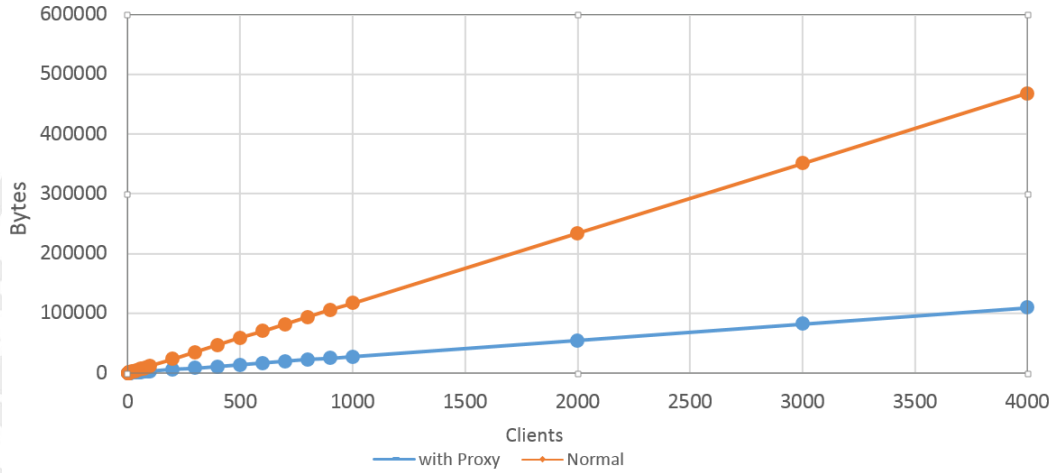


Figure 5.10: Bytes need to perform a Write operation with Payload of 5 Bytes

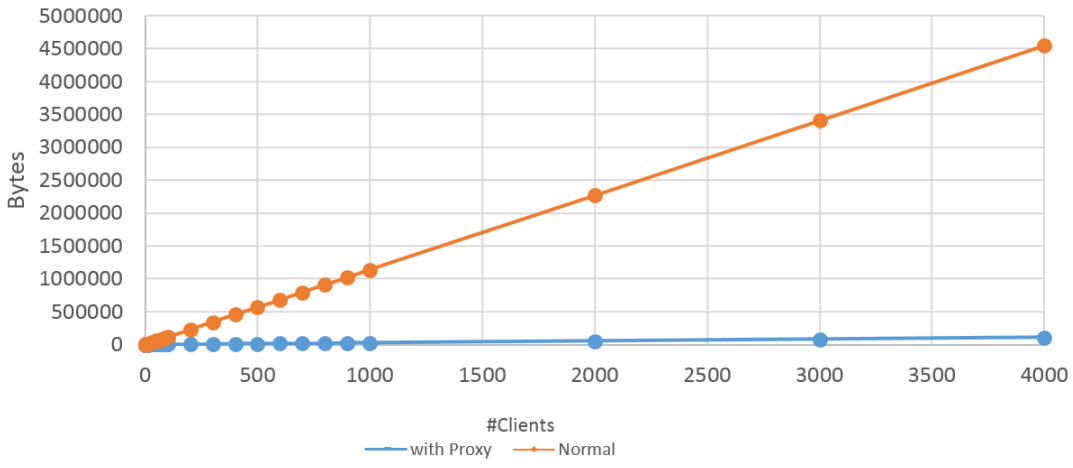


Figure 5.11: Bytes need to perform a Write operation with Payload of 1024 Bytes

The results, showed in Figures 5.10 and 5.11, prove clearly that the solution with Proxy has less overhead than the unicast solution. This is true for a number of clients in the group that is higher than 4 (in case payload length is 5), and already for group of only two members in case the payload length is 1024.

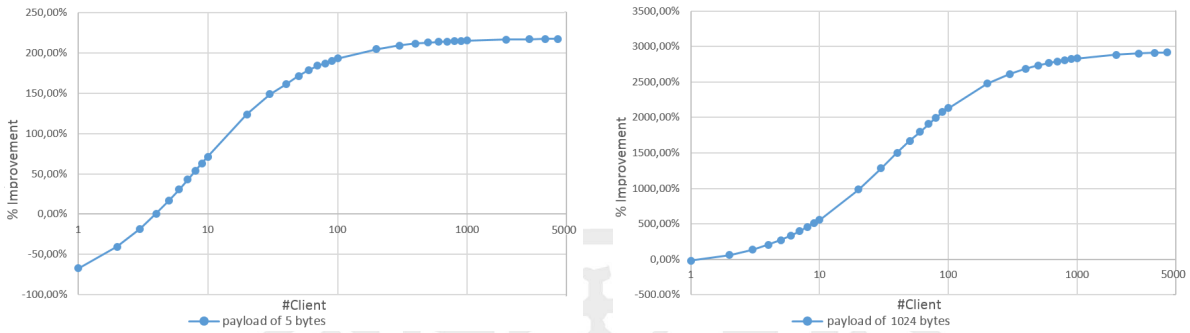


Figure 5.12: Improvement Provided by the proposed solution for Write Operation

Figure 5.12 show that the improvement for payload of 5 bytes there is already a benefit around 75% for 10 clients, which goes up to over 220% for bigger number of clients. For payload of 1024 bytes instead, the advantage is of over 500% for a group of 10 member that goes up to almost 3000% if the number of members is over 1000. These result were to be expected since in the proposed solution the payload is sent only a single time, while in the unicast is sent every time for every client. Therefore the Group Write operation is very advantageous an should be adopted as main option instead of using unicast messages.

Observe

Regarding Observe operation, in a Server-Client system once the observe request is performed, the clients periodically sends updated value of the resource observed. This operation requires the same 57 bytes plus for the request and $57 + P_{len}$ bytes for the Response, of which P_{len} is the payload.

In the Server-Proxy-Client system instead, we stated in Section 4.6, that once the Proxy receives the *Create* request for the GroupObserve Object and eventually the Observe request for the *value* resource (that stores the aggregated Notify) it waits for a Notify from all members (or timeout) and then sends the aggregated Notify back to the Server. This implies that bytes required to start a Group Observation and obtain the first Notify is given by the following formula

$$CreateGroupReq + Ack + ObsReq + (NotifyHeader + CPayload) * NClient \quad (5.3)$$

Where $CreateGroupReq$ is the size of the *Create* Request and requires 57 bytes and $Args$ is $20 * NClient$ (20 is our average length of a Client ID/UIID, but could be smaller, like we said in 3.5.4). 57 bytes are required for $ObsReq$, the Observation request for the *value* resource, then we have the $NotifyHeader$ that may change based on what format we use for reporting information. In fact, since the *value* resource is in truth of type *Multiple Resource*, we have the opportunity to use JSON or TLV as payload format ([3]), thus we conducted the test for both cases.

JSON Results of test conducted using JSON as response format are showed in Figure 5.13. We tested for both a Client Payload of 5 bytes, which should be the average response value, and 250 bytes.

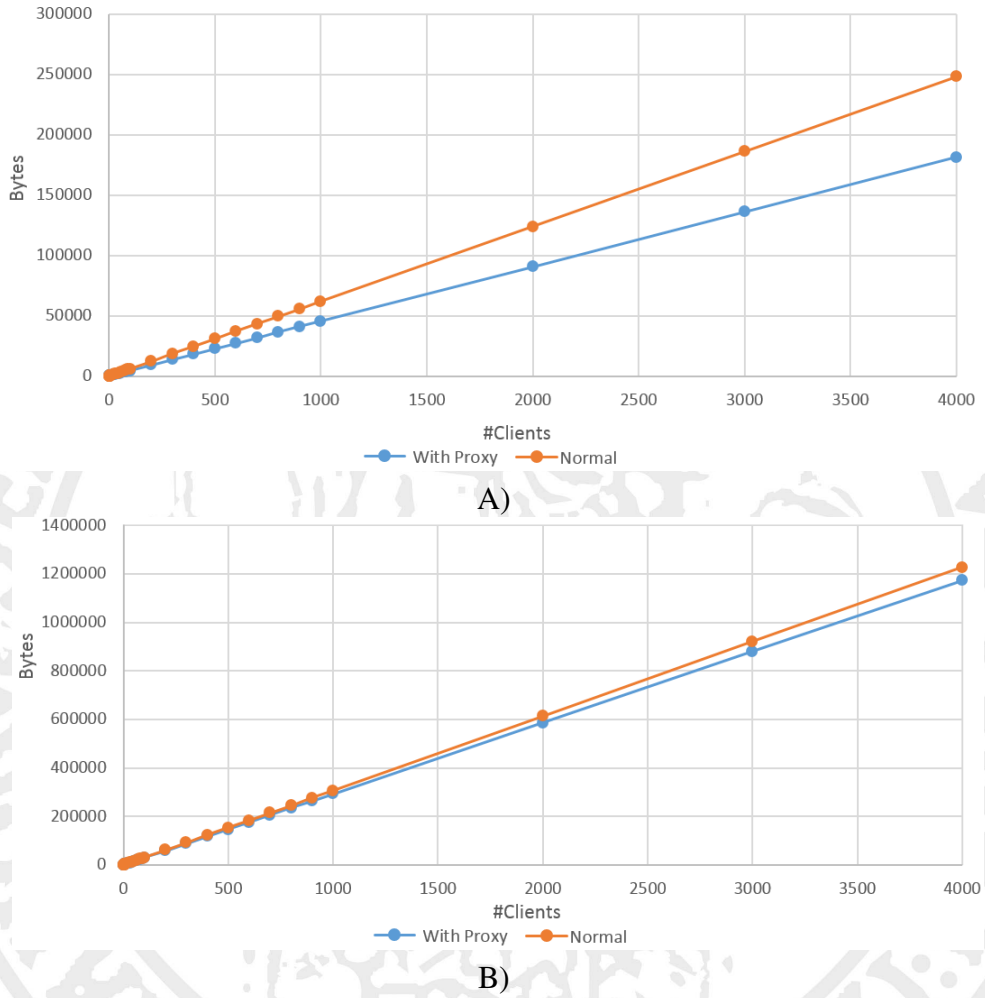


Figure 5.13: Bytes need to perform an Observe operation with Payload of 5 Bytes (A) and 250 Bytes (B)

As results show for both cases the Proxy solution provides better results than the unicast solution, even if by a slight margin with larger ClientPayload.

The difference can be appreciated more in detail in Figure 5.14. As we can see, the improvement provided in for payload of 5 bytes long is just slightly positive, but only if the number of group members is greater than ten, otherwise is very disadvantageous.

Slightly better results are obtained in the case of payload of 250 bytes long, but here also the number of group members must be greater than ten. These result lead us to think that the JSON format should be avoided if non really needed (for example for parsing reason), this decision looks even more reasonable if we look at the performance of the TLV format.

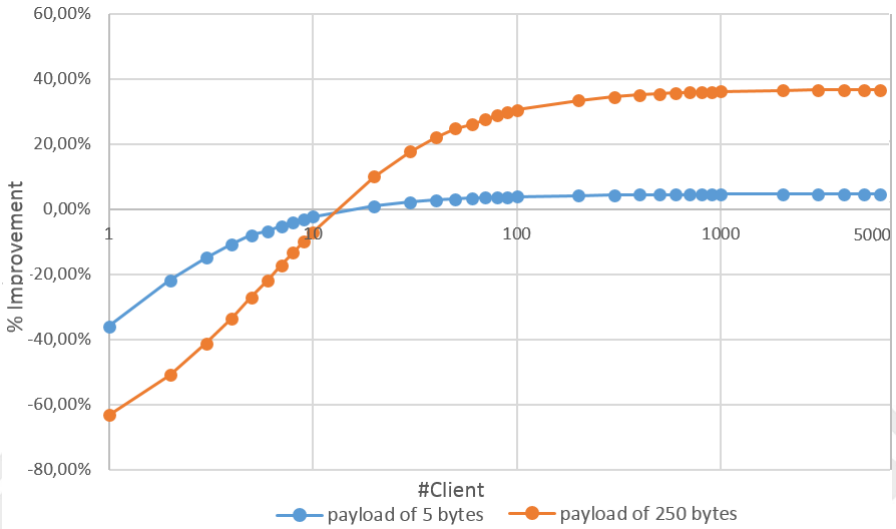
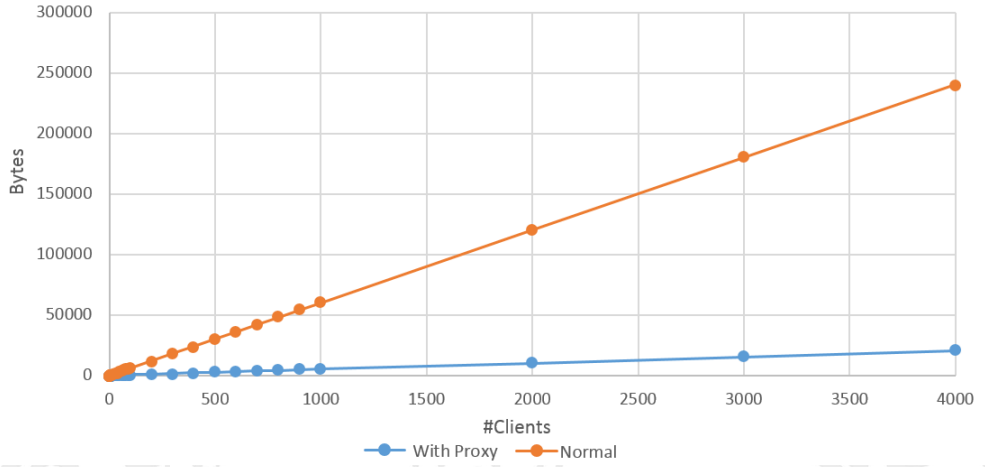


Figure 5.14: Improvement Provided by the proposed solution for Observe Operation

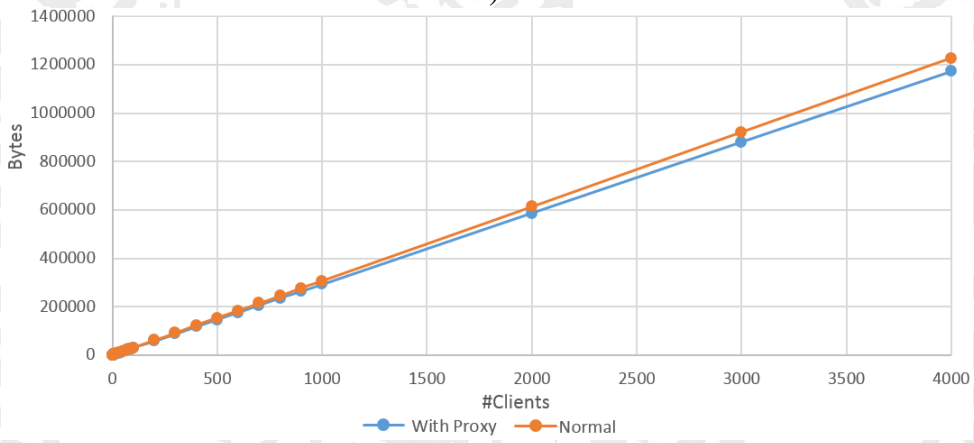
TLV Results of test conducted using TLV as response format are showed in Figure 5.15. As done with the JSON format, we tested for both a Client Payload of 5 bytes and 250 bytes. In the case of payload length of 5 bytes, there is an huge difference in terms of byte used form our solution with the normal unicast solution. While these differences almost disappear in case of Payload length of 250 bytes, but in both cases our solution is still advantageous in term of bytes consumed.

The big difference of the two results is shown in Figure 5.16. The test cases where the payload length was 5, produces an increment in performance of almost 800% and this result is obtained for group where the number of member is greater than 5. In the case of payload length of 250, the improvement obtained is just slightly positive. This is due to fact that the payload size of every client is a lot bigger than the overhead (sum of the header of every layers) needed to ship the payload to destination.

The Results obtained lead us to think that the TLV format should be used in response (Notify) of Group Observe operation. In this way is possible to achieve the best result, independently from the payload length but must be taken in consideration the number of group members should be at least bigger than five in order to always achieve a positive result.



A)



B)

Figure 5.15: Bytes need to perform an Observe operation with Payload of 5 Bytes (Left) and 250 Bytes (Right)

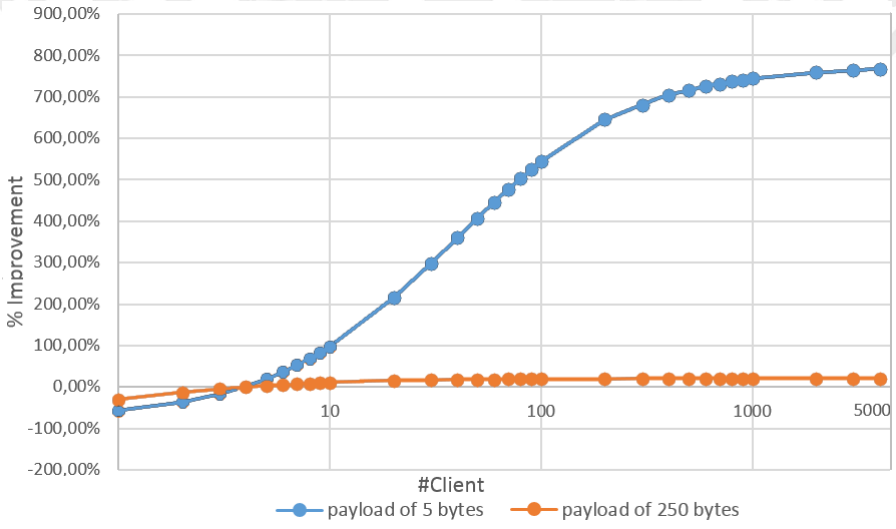


Figure 5.16: Improvement Provided by the proposed solution for Observe Operation

Chapter 6

Conclusion and Future Work

This thesis work was part of the work conducted at NomadicLab (Ericsson Finland) in order to find solutions to improve managements of M2M systems using LWM2M protocol. The solution we proposed was the introduction of Group Communication Service for LWM2M protocols in order to manage multiple device by assembling them in group and optimizing operation on them. We close the thesis with e recap of what was done in this project, our conclusions and what future work is needed.

6.1 Summary

In this thesis we studied how to optimize and improve LWM2M operation for a big number of devices. The solution, for which we also implemented a prototype, focused on a Group Communication Service for LWM2M which provides group management, aggregation and network optimization.

To introduce group communication mechanisms we choose multicast model, which has the plain advantage of bandwidth efficiency (extremely important when the number of devices is really high), and in particular we choose Application Layer Multicast (ALM) due to the impossibility to guarantee IP multicast in every segment of the network (see section 2.2.3).

Having chosen an ALM solution we needed to build an API that would satisfy common requirements of group communication (listed in section 3.2), like: discovery, creation and removal of groups; discovery, addition and removal of group members; and try to perform LWM2M operations on groups in the same way a LWM2M Server perform operations on a LWM2M Client. Keeping in mind principles like interoperability, scalability, security, and reliability, the design

phase resulted in the creation of a LWM2M Proxy, meant to act as intermediary between Server and Client, and the decision to represent Groups as LWM2M Object Instances.

The LWM2M Proxy is both a Server and Clients, thus provides both interfaces allowing to receive commands (addressing a particular group) from Server and forward them to group members. the Proxy was also designed with the purpose of providing Group mechanism through its own LWM2M Objects. Objects that, since it is has a Client-side, the Server will only need to address a normal Client to use functionality they provide.

The decision to use an object as Group allowed to create, modify and delete Groups as Object instances with normal LWM2M commands, but the different nature of LWM2M interfaces, and thus of the operation to be mapped, required to use more than one object for mapping. In fact the *Device Management* (DM) interface implements a simple server-client pattern, while the *Information reporting* (IR) interface implements a publish/subscribe pattern. For this reason we decided to design a set of objects template to be used as group representative while others were designed to provide support for group mechanisms.

We called the two objects used to map LWM2M interfaces *GroupObject*, for DM interface, and the *GroupObserve*, for IR interface. Both objects template were designed with executable resources to add, edits or remove group members and other resource that allowed to perform the group operation for the interface they map.

The Group object was designed with a *Forward* resource that when executed, it forwards DM commands provided as arguments from the Server to the group members, then it stores an aggregate response (that includes the response from every member) in the *FW response* resource.

The GroupObserve Object instead, was designed assuming that the member of the Group was not the client but the resource to be observed. For this reason, when a GroupObserve Instance is created, the proxy start an *Observation* on all the resources indicated as member of the group and the response/notification are stored in the *Values* resource of the instance.

With this structure, other properties of the group/object instance can be easily accessed as Resource of the object. Examples are the Id of the Group or the list of members (*GroupId* and *Members* Resources) which are available in both Group Objects.
Other objects were designed in order to provide support functionality useful for grouping mechanism and management: *RegistrationPolicy* Object provides automatic group creation through a policy system that check if a newly registered Client has some properties (resource values) that satisfy all the conditions defined in the policy*; Observed Monitor* helps to track which resources are currently observed and from who; *ProxiedDevices* helps a Server to know which devices are

registered to the Proxy, providing the server all the information needed to be able to manage the clients and to create new groups.

Based on our design, a prototype of the Proxy and its object was implemented. The prototype was developed using Wakaama, an implementation in C of LWM2M [44], and its Client and Server Example. We explained what choice we made during implementation of the prototype keeping in mind some of the future development (explained next) and finally tested the proper working of our design.

Lastly, we performed some basic testing on memory usage of the Proxy and compared the standard Server-Client system and our Server-Proxy-Clients systems, measuring the overhead and network network traffic generated to perform common LWM2M operations on an increasing number of Clients and between different response formats. Results showed that our prototype as it is already provide an increments in performance of the systems and thus could be used in practical scenarios.

6.2 Future Work

In the developing of this work, we focused on base functionality of Group Communication Mechanism, making some assumption and ignoring some particular use cases that should be managed differently. Two point that should be analyzed and improved are the *security* and the *queue mode*.

Regarding security, we assumed that the proxy uses the same Control Access provided by LWM2M. This imply that if a LWM2M Server has access right to the LWM2M Proxy and its Objects, the Server is automatically able to perform every kind of commands on all the LWM2M Clients registered to the Proxy.
This behavior it is not always advisable and should be addressed. One soultion is to copy Clients' ACL in the Proxy, enabling the Proxy to check if the Server is autorized to perform the operation. Alternatively, the Server could sends an auth Token with the command to be forwarded, allowing clients to perfrom a standard access control. In both cases, the Proxy need to be improved.

Regarding the Queue Mode, LWM2M states that the server must not send immediately a request, but wait for the client to be available and then send all request *in queue* together. This mode obviously collides with the current behavior of our Prototype where the aggregate response sent to the server have the response code set as "Server Unavailable". For this reason a solution to cope with this case should be developed.

Lastly, even if our prototype showed good results regarding performance, further measurement should be performed with the addition of simple improvement like a caching functionality and merging of duplicate commands from multiple servers or measurement with different undunderlying protocol from the perspective of latency and throughput in real-time systems, since they have special requirement for those parameters.

Aknowledgement

There are many people that I have to thank for helping me to complete this thesis work.

First, I have to thank Professor Simon Pietro Romano, Supervisor of this thesis, for his guidance and assistance through this project.

I would like to thank peoples at Ericsson Research NomadicLab, where this work was carried out. They were very friendly and they made me feel welcomed. Special thanks to Jaime Jiménez Bolonio for his advices in the design and implementation phases of the systems and for his useful comments on my thesis drafts. Thanks also to Mert, Oscar, Aimmy, Miika, Roberto, Ines for the good time we spent together, especially at lunch.

Also I have to thank every friends met in my university studies for having shared that experience with me. You are too many to be listed here but you should know that every one of you influenced my someway

Thank to Vania, for her being so inspiring every time I had to find the best solution for my work, even if she did not have the slightest idea of what I was talking about.

At last I would like to thank my parents. They motivated me, were always supportive and they showed me that hard times can always be overcomed, you just have to want to. Thank you.

Bibliography

[1] REST - Representational State Transfer - Les concepts.

[2] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252 (Proposed Standard), June 2014.

[3] Open Mobile Alliance. Lightweight machine to machine technical specification. *OMA-TS-LightWeightM2M-V1_0-20131210-C, Candidate version*, 1(10), 2013.

[4] Li Lao, Jun-Hong Cui, Mario Gerla, and Dario Maggiorini. A comparative study of multicast protocols: top, bottom, or in the middle? In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 4, pages 2809–2814. IEEE, 2005.

[5] Sushant Gupta and Ankit Hirdesh. Overview of m2m. *Ankit Hirdesh Papers website: http://hriday. ankit. googlepages. com/M2M_overview_paper. pdf*, 2007.

[6] Kamal Moummadi, Rachida Abidar, and Hicham Medromi. Distributed resource allocation: Generic model and solution based on constraint programming and multi-agent system for machine to machine services. *International Journal of Mobile Computing and Multimedia Communications (IJMCMC)*, 4(2):49–62, 2012.

[7] Chanakya Kumar and Rajeev Paulus. A prospective towards m2m communication.

[8] INFSO D.4 NETWORKED ENTERPRISE & RFID INFSO G.2 MICRO & NANOSYSTEMS. Internet of things in 2020 - roadmap for the future.

[9] Dag Björklund Johan Westö. An overview of enabling technologies for the internet of things. *website: https://eva.fing.edu.uy/pluginfile.php/76857/mod_folder/content/0/IoT.pdf*, 2014.

[10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[11] Gilles Privat. From smart devices to ambient communication. *Research & Development*, 2006.

[12] LM Ericsson. More than 50 billion connected devices, 2011.

[13] Wim Elfrink John Chambers. Cisco ceo: Why the future of the internet is already here. *website: http://fortune.com/2014/07/16/cisco-ceo-why-the-future-of-the-internet-is-already-here/*, July 2014.

[14] Cisco Karen Tillman. How many internet connections are in the world? right. now, 2013.

[15] K.R. Rao, Z.S. Bojkovic, and B.M. Bakmaz. *Wireless Multimedia Communication Systems: Design, Analysis, and Implementation*. Taylor & Francis, 2014.

[16] Geng Wu, Shilpa Talwar, Kerstin Johnsson, Nageen Himayat, and Kevin D Johnson. M2m: From mobile to embedded internet. *Communications Magazine, IEEE*, 49(4):36–43, 2011.

[17] TS ETSI. 102 689 m2m service requirements.

[18] Suman Pandey, Mi-Jung Choi, Myung-Sup Kim, and James W Hong. Towards management of machine to machine networks. In *Network Operations and Management Symposium (APNOMS), 2011 13th Asia-Pacific*, pages 1–7. IEEE, 2011.

[19] Roy Fielding. Fielding dissertation: Chapter 5: Representational state transfer (rest), 2000.

[20] R. Hill, L. Hirsch, P. Lake, and S. Moshiri. *Guide to Cloud Computing: Principles and Practice*. Computer Communications and Networks. Springer, 2012.

[21] Nilo Mitra, Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. *{SOAP} Version 1.2*. World Wide Web Consortium, June 2003.

[22] onem2m. Architecture analysis - part 2: Study for the merging of architectures proposed for consideration by onem2m, 2013.

[23] Christophe Diot, Brian Neil Levine, Bryan Lyles, Hassan Kassem, and Doug Balensiefen. Deployment issues for the ip multicast service and architecture. *Network, IEEE*, 14(1):78–88, 2000.

[24] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures, University of California, Irvine, 2000*, chapter Representational State Transfer (REST). Dissertation, 2011.

[25] S. Krco Z. Shelby, C. Bormann. Core resource directory. *website: http://tools.ietf.org/html/draft-ietf-core-resource-directory-01*, 2014.

[26] Z. Shelby. Constrained RESTful Environments (CoRE) Link Format. RFC 6690 (Proposed Standard), August 2012.

[27] Hartke andU niversitaet Bremen TZI. Observing resources in coap. *draft-ietf-core-observe-14 (work in progress)*, 2014.

[28] Brad Hale. network management system. http://whatis.techtarget.com/definition/network-management-system.

[29] J.D. Case, M. Fedor, M.L. Schoffstall, and J. Davin. Simple Network Management Protocol (SNMP). RFC 1157 (Historic), May 1990.

[30] Douglas Mauro and Kevin Schmidt. *Essential snmp*. " O'Reilly Media, Inc.", 2005.

[31] K. McCloghrie and M. Rose. Management Information Base for Network Management of TCP/IP-based internets:MIB-II. RFC 1213 (INTERNET STANDARD), March 1991. Updated by RFCs 2011, 2012, 2013.

[32] DSLHome-Technical Working Group et al. Tr-069 cpe wan management protocol. In *Broadband Forum*, 2004.

[33] Broadband Forum - http://www.broadband-forum.org.

[34] IPSO Alliance. Enabling the internet of things. *IPSO Alliance.[Internet] Available at: http://ipso-alliance. org.[Accessed 18 August 2011]*, 2011.

[35] Internet Protocol for Smart Objects (IPSO) Alliance. IPSO Smart Objects,Smart Objects Starter Pack 1.0". https://github.com/connectIOT/lwm2m-objects/tree/master/ipso/1.0.

[36] IPSO OMA. Omna lightweight m2m (lwm2m) object and resource registry. *website: http://technical.openmobilealliance.org/Technical/technical-information/omna/lightweight-m2m-lwm2m-object-registry*, 2014.

[37] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), January 2012.

[38] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008. Updated by RFC 6818.

[39] Akbar Rahman and Esko Dijk. Group communication for coap. *Group*, 2013.

[40] E Dijk and A Rahman. Miscellaneous coap group communication topics. *draft-dijk-core-groupcomm-misc-06 (work in progress)*, 2014.

[41] Mojtaba Hosseini, Dewan Tanvir Ahmed, Shervin Shirmohammadi, and Nicolas D Georganas. A survey of application-layer multicast protocols. *IEEE Communications Surveys and Tutorials*, 9(1-4):58–74, 2007.

[42] et al P. Leach. Uniform Resource Identifiers (URI): Generic Syntax. RFC 4122 (Proposed Standard), 2005.

[43] Intel Open Source Technology Center (01.org). LibLWM2M. https://github.com/01org/liblwm2m.

[44] The Eclipse Foundation. Wakaama. http://projects.eclipse.org/projects/technology.wakaama.